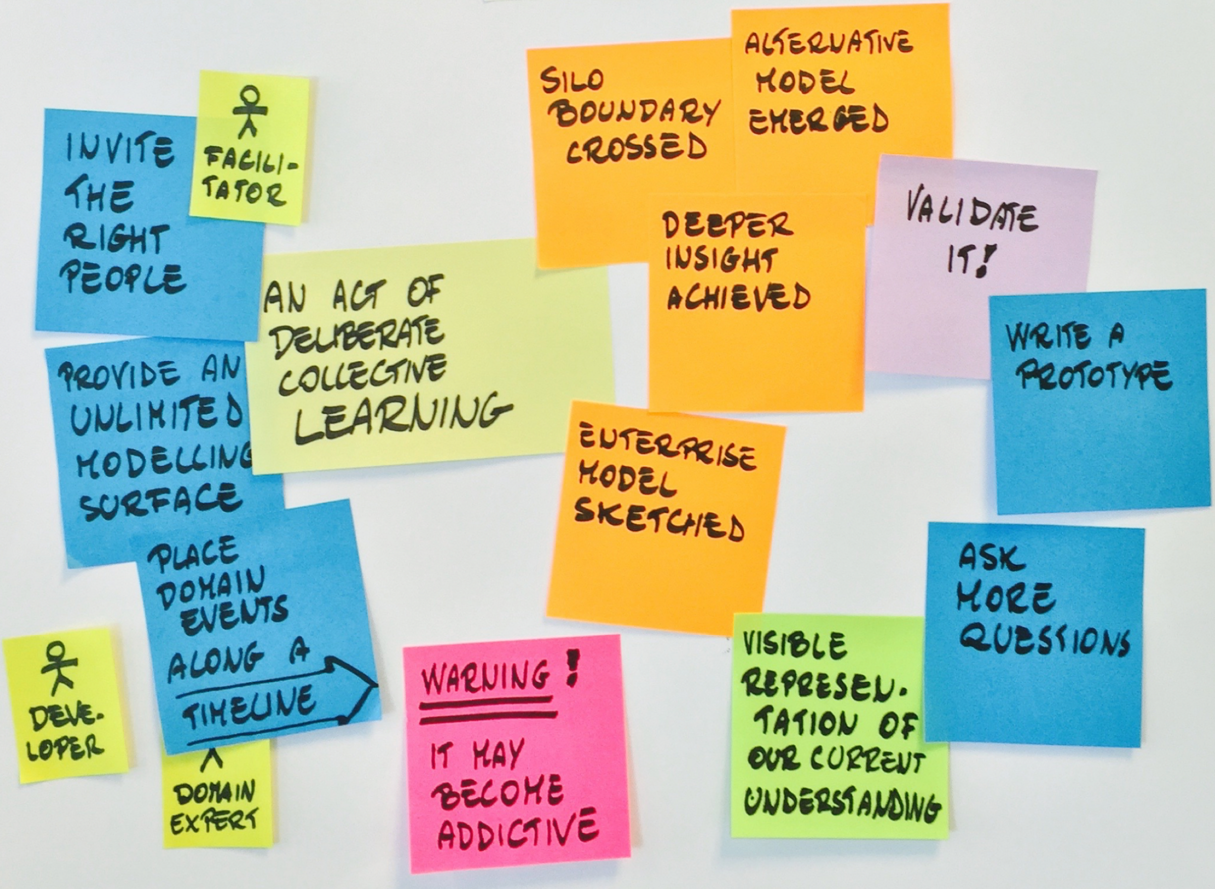
EVENT STORMING

Alberto Brandolini


INVITE THE RIGHT PEOPLE
FACILI-TATOR
PROVIDE AN UNLIMITED MODELLING SURFACE
AN ACT OF DELIBERATE COLLECTIVE LEARNING
PLACE DOMAIN EVENTS ALONG A TIMELINE
DEVE-LOPER
DOMAIN EXPERT
SILO BOUNDARY CROSSED
ALTERNATIVE MODEL EMERGED
DEEPER INSIGHT ACHIEVED
VALIDATE IT!
WRITE A PROTOTYPE
ENTERPRISE MODEL SKETCHED
WARNING! IT MAY BECOME ADDICTIVE
VISIBLE REPRESEN-TATION OF OUR CURRENT UNDERSTANDING
ASK MORE QUESTIONS

Introducing EventStorming

An act of Deliberate Collective Learning

Alberto Brandolini

This book is for sale at http://leanpub.com/introducing_eventstorming

This version was published on 2018-06-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2018 Alberto Brandolini

Contents

Preface - 25% . . . . **i**
About this book i
Who is this book for i
Notation ii
Acknowledgments ii
How to read this book ii
Version iii

1. What does EventStorming look like? - 85% **1**
Challenging corporate processes 1
Kicking off a startup 7
Designing a new feature for a web app 18
Quick EventStorming in Avanscoperta 21

A deep dive into problem space **26**

2. A closer look to the problem space - 80% **27**
Complexity in a nutshell, or less 27
Organization silos 35
Hierarchy 43
The shape of the problem 44

3. Pretending to solve the problem writing software - 50% **47**
It’s not about ‘delivering software’ 47
The illusion of the underlying model 51

The Product Owner fallacy 55
The backlog fallacy [FIXME: definitely not the first one 56
The backlog fallacy (rewritten) 64
Modeling is broken 65
Requirements gathering is broken 66
Enterprise Architecture is broken 66
The EventStorming approach 67

4. Running a Big Picture Workshop - 98% 68
Invite the right people 70
Room setup 70
Workshop Structure 72
Phase: Kick-off 73
Phase: Chaotic Exploration 75
Phase: Enforcing the timeline 80
People and Systems 88
Phase: Problems and opportunities 100
Phase: Pick your problem 102
The promised structure summary 106

5. Playing with value - part 1 108
Explore Value 108
Explore Purpose 108

6. Making it happen 109
Managing Participant’s experience 109
Managing conflicts 114

7. Preparing the workshop - 30% 116
Choosing a suitable room 116
Provide an unlimited modeling surface 119
Managing invitations 123

8. Workshop Aftermath -20% 129
Cooperating domains 129
When to stop? 129

How do we know we did a good job? 129
Wrapping up a big picture workshop 132
Managing the big picture artifact 133
Focusing on the hot spot 133
Documenting the outcomes - TRICKY 135
Emerging structure 135

9. Big Picture Variations - 50% **137**
Software Project Discovery 137
Organization Retrospective 138
Induction for new hires 138

Why is it working? **140**

10. What software development really is - 40% **141**
Software development is writing code 141
Software development is learning 142
Software development is making decisions 144
Software development is waiting 145

11. Learning is the problem, so what? - 20% **146**
Understanding our ignorance 146
How does learning happen? 146
The Dreyfuss model as a constraint for learning 146
The invisible deliverable 147
Learning without action is waste 147

People Interaction **149**

12. People In Action - 25% **150**
Managing Invitations 150
Igniting the workshop 151
Mastering Workshop Dynamics 153
Observing Body Language 155

Enforcing narratives 157
Walking Skeleton - redefined 159
Is it really a failure? 159
Nobody likes to be wrong 160
Brainstorming biases 160

13. facilitating the workshop - 10% 161
The economics of a meeting with the big guys 161
Promote questions 162
Allow people to checkout 162
Allow people to stay 162
Keep your mouth shut 162
Introverted and extroverted 162

14. Breaking the space constraint - 30% 163
Overcoming your whiteboard implicit constraint 163
The illusion of the unlimited model 165

Modeling processes and services 167

15. A platform for cooperation 168

16. Exploring value 169
Explore Value Created and Destroyed 169
Explore multiple currencies 171
What to do when there is no value? 171
Look for inconsistencies 171

17. Observing global state - 10% 172
The transaction obsession 172
There's more to consistency than it's apparent to the eye 174

Modeling software systems **175**

18. Running a Design-Level EventStorming - 10% **176**
Scope is different 176
People are different 176
What do we do with the Big Picture Artifact? 177
Where are Events Coming from? 178
Discover Aggregates 179
How do we know we're over? 180

19. The picture that explains everything - 5% **181**

20. Three types of blockers **183**
User Decisions 183
Processes and policies 184
What happened to aggregates? 184

21. Design-Level modeling tips **185**
Make the alternatives visible 185
Choose later 185
Pick a Problem 185
Rewrite, then rewrite, then rewrite again. 186
Hide unnecessary complexity 186
Postpone aggregate naming 187

22. Building Blocks - 20% **188**
Why are Domain Events so special? 188
Events are precise 190
No implicit scope limitation 190
Domain Events as state transitions 192
Domain Events are triggers for consequences 192
Domain Events are leading us towards the bottleneck 193
Alternative approaches 193
Wrapping everything up 193
Commands - Actions - Decisions 194

23. Color Modeling Patterns - 10% **195**
Policies 195
Why Orange? 201
Time Triggered Events 203

24. Modeling Aggregates **205**
Discovering aggregates 206

25. Event Design Patterns - 5% **208**
Discovery strategies 208
Composite Domain Event 208

26. Collaborative modeling tips and tricks **210**
Model the right problem 210
Keep the model clean 210
Postpone branches 210
Finish First / Correct Later 211
Modeling out loud 211
Brainstorm on terms 211
Brainstorm on context 211
Summary 211

27. From paper roll to working code **212**
Managing the design level EventStorming artifact 212
Coding ASAP 213

28. From EventStorming to UserStories - 5% **214**
A placeholder and a better conversation 214
Defining the acceptance criteria 214
EventStorming and User Story Mapping 214
How to combine the two approaches? 215

29. Modeling in the large - 10% **217**
Enter Bounded Contexts 217
Bounded Contexts modeling patterns 217

30. Working with Startups - 2% **220**

The focus is not on the app 221
Leverage Wisdom of the crowd 221
Multiple business models 221

31. Working in corporate environment - 5% 222
Invitation is crucial 222
Manage check-in process 224
The fog-me-fog model 224
Nobody wants to look stupid 226
Wrapping up 226
What happens next? 226
Corporate Dysfuctions 226

32. Designing a product 228
This is not a tailored solution 228
Matching expectations 228
Simplicity on the outside 228

33. Model Storming - 0% 229

34. Friends and ncestors 230
Ancestor and relatives 230

35. Remote Event Storming 233
Ok, seriously 234
Downgrading expectations 235

Patterns and Anti-patterns 236

36. Patterns and Anti-Patterns - 75% 237
Add more space 237
Be the worst 238
Conquer First, Divide Later 239
Do First, Explain Later 240
Fuzzy Definitions 241

Enforce the Timeline 242
Guess First 242
hot spot 243
Icebreaker (the) 243
Incremental Notation 244
Go personal 244
Keep your mouth shut 245
Leave Stuff Around 245
Manage Energy 246
Make some noise! 246
Mark hot spots 246
Money on the table 247
One Man One Marker 247
Poisonous Seats 248
Reverse Narrative 248
The Right To Be Wrong 249
Single Out the Alpha-male 249
Slack day after 249
Speaking out loud 250
Start from the center 250
Start from the extremes 251
Unlimited Modeling Surface 251
Visible Legend 251

37. Anti-patterns 253
Ask Questions First 253
Big Table at the center of the room 254
Committee 255
Divide and Conquer 256
Do the right thing 257
Dungeon Master 257
Follow the leader 257
Human Bottleneck 257
Karaoke Singer 257
Precise Notation 258
Religion War 259

The Spoiler 261
Start from the beginning 262
The godfather 262

38. RED ZONE 263
Fresh Catering 263
Providential Toilet Door Malfunctioning 263

Specific Formats 264

39. Big Picture EventStorming 265

40. Design-Level EventStorming 267
Next actions 268

41. Value Stream EventStorming 269

Glossary - 40% (but do you really care?) 270
Fuzzy by design 270

Tools 274
Modeling Surfaces 275
Markers 276
Stickies 277
Static pads 279
Recording results 280

Bibliography 281

Preface - 25%

The reason why I'm writing

About this book

I've spent a lot of time trying to understand what was the best scope for this book. From an explorer perspective, EventStorming is great fun: every workshop I run gives me the opportunity to discover new variations and new opportunities to learn interesting things in unexpected ways. But from a book author perspective this was slowly turning into a *nightmare*: I was discovering things faster than my ability to write them down, the estimated completion date for a comprehensive EventStorming book kept moving further into the distant future.

At the same time, many of the discoveries along the way had little to do with the original idea, and probably weren't that interesting for some of the people that already gathered around it.

So this LeanPub book is my *Minimum Viable Product*: an attempt to get in touch with the first community of readers, to gather valuable feedback and write a good book without getting trapped into the "second edition" approach.

Who is this book for

I've realized that the only way to actually deliver something was to start narrowing down the audience, and write to a specific community.

So this book is for the early practitioners and for those that found the idea of EventStorming appealing for improve the learning around software development process. That's not the only community that will benefit from EventStorming, but it's the first one that's badly *asking* for it. So here we are!

Notation

I'll be strict in being consistent with my own notation. This means that I'll be using a rigid color scheme (orange for *Domain Events*, purple for *Hot Spots*, blue for *Commands* and so on).

It's not necessarily the best color choice. Some pointed out that it's not that easy to purchase the required amount of orange stickies in regular shops, especially if there's another eventstormer in town.

Be free to chose yours, I kept using the original one because it had a rationale, and because it makes every picture of real workshop easily intelligible.

Acknowledgments

[FIXME: this is going to be a big mess. So many people to thank…]

How to read this book

This book is not finished. It's actually nowhere close to being finished. But I committed myself entirely in finishing it. Even worse: I have a structure, but I find it very hard to follow a sequence when I am writing. What I write is the answer to a question in my head (maybe due to a recent conversation), that deserves a place in the existing structure, or maybe destroys it.

Progress counter on top of chapters

I've added a progress indicator in the chapter headers, in order to have a feeling of the chapter status before reading it. Of course the counter is arbitrary, but it may be useful to some.

Dense content

Actually, the biggest hole is in the middle. If you're already into EventStorming you'll probably find the more interesting content in the Patterns and Anti-Patterns sections, with a collection of self-contained tips and tricks.

Progress

If you're interested in seeing progress for this book, I made my personal reward generating spreadsheet visible[1].

Feedback

There are a few ways to give feedback. Some folks already discovered Twitter as a rapid way, but that would probably drive me nuts (you can actually run a Denial of Service attack on me. It's probably fun).

The official feedback page[2] is on LeanPub.

I've also set up a public Trello board[3] for more transparent strategic feedback, progress and discussion.

Version

Main version is 0.1.0.2

---

[1] https://docs.google.com/spreadsheets/d/1kk9Cfe6AF6keVN8VBOCdXe3VZ_2QfON8SvtozlOIcos/edit?usp=sharing

[2] https://leanpub.com/introducing_eventstorming/feedback

[3] https://trello.com/b/tcLNC1sG/eventstorming-book

1. What does EventStorming look like? - 85%

You may have heard about EventStorming, this odd-looking workshop that has a massive consumption of orange sticky notes. You may have even practiced it with some colleagues in your company. And maybe you are reading this book looking for guidance about what's the best way to run an EventStorming workshop.

It's only the second paragraph and I am already sorry to disappoint you: there is no *'best way'*.

In fact, there are many different ways to run an EventStorming, with different purposes and rules, but with a few things in common.

Here you'll have a taste of what you might expect, with four different EventStorming stories.

Challenging corporate processes

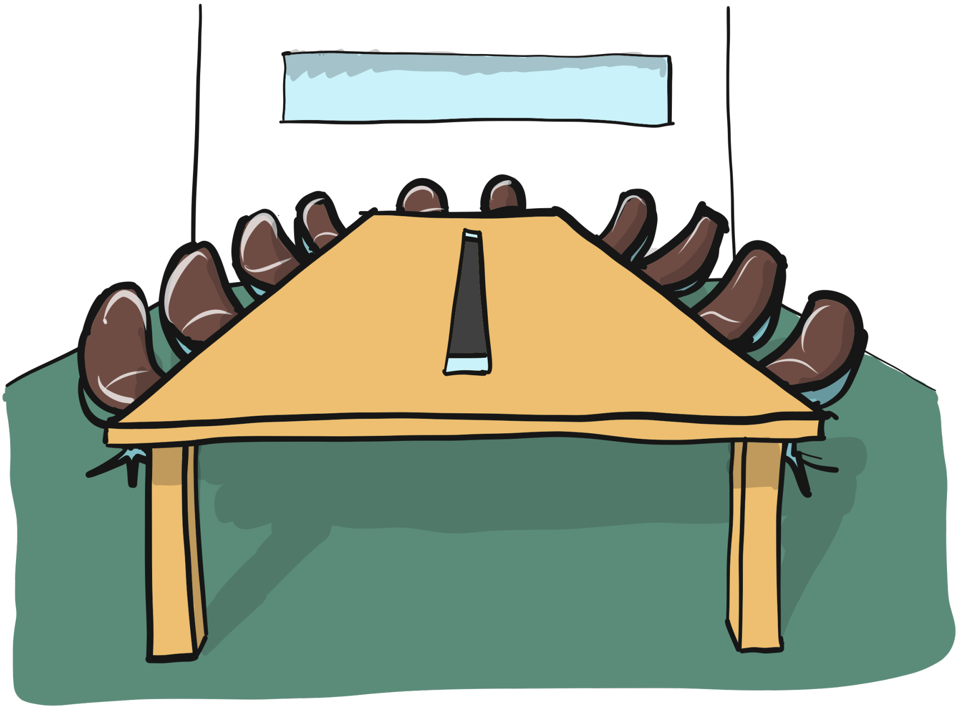
A mid-size company is struggling with a typical growth problem: the IT infrastructure that used to provide an edge in the early days is now slowly becoming a burden, turning daily operations into a pain and ultimately making some strategic growth options non-viable.

I am invited to provide some big-picture level assessment of the state of the art in order to highlight the next critical actions.

The workshop

The situation looks so compromised we can't even manage the invitation process right: some key people declared they won't show up. However, we need to start from somewhere. We agree to call the meeting anyway, with the possibility of a follow-up.

The room we're given looks like the typical corporate meeting room: a large table in the center, and a dozen of chairs around it. Poisonous[1].



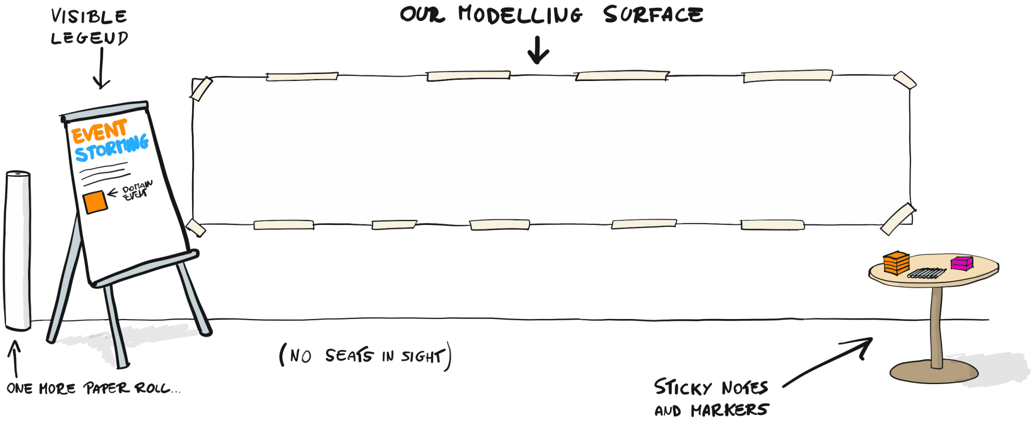
Nothing smart will ever come out from this setting, but …please, take your seat!

We take control of the room 30 minutes before participants are expected to arrive, in order to hack the space in our favor. We push all the chairs to the corners and move the table to the side (it's one of those huge meeting room tables, that made you wonder *"How the hell did they bring it in?"*), and we stick

---

[1] If you want to know why, have a look to Big table at the center of the room in the Anti-patterns section.

8-9 meters of a plotter paper roll onto the main wall. Then I extract a dozen of black markers from my bag, and a stockpile of sticky notes, mostly orange.

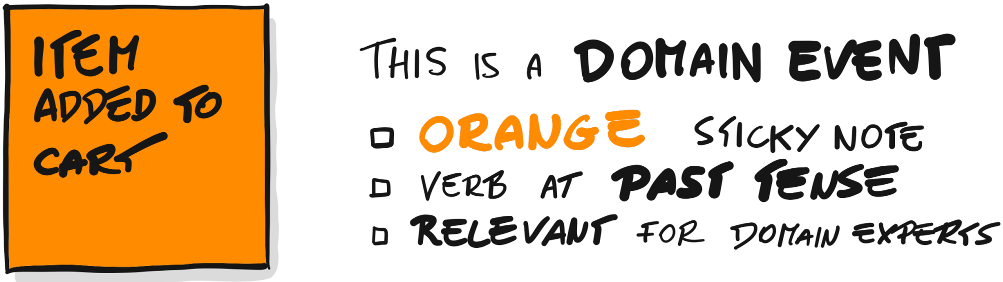


The room setup, right before the workshop

Eventually, participants - representatives from the different company departments, plus the IT - start to show up. Some look puzzled at the unusual room setup, desperately looking for a seat. Someone else is still missing but we decide to start.

I kick off the meeting with a short introduction: *"we're asking you to write the key events in your domain as an orange sticky note, in a verb at past tense, and place them along a timeline"*.

We decide to narrow the scope only to the process of customer acquisition, since it looked big enough, given the time constraints (2-3 hours overall).



All you need to know about a Domain Event to get you started

Some participants still look confused, but the majority starts writing their own stickies and placing them on the paper roll. The *icebreakers* are doing their job brilliantly, acting as examples for the colleagues, that quickly begin to imitate them. This way, we're getting into a state of *flow* in a few minutes.

I offer my help to the ones that still look uncomfortable, answering questions or providing a working marker if they're missing one. It turns out that I misunderstood their disengagement: *"I was hired two weeks ago. I am finally understanding what this company is doing!"*, a person whispers.

The whole process starts to make sense, and while we're merging the different points of view, people can't stop commenting on given business steps: *"and this is where everything screws up!"* or *"this never really works as expected"* or *"it always takes ages to complete this step"*.

This information is crucial, so I try to capture every warning sign and write them on purple stickies decorating them with big exclamation marks, then I place them close to the corresponding step of the emerging workflow.

Capture warnings, discussion and question marks along the way

As we expected, the number of critical issues is significant. But placing them close to the corresponding events gives a hint about where we should concentrate our efforts.

Even my mood is different, now. I asked very beginner's questions at the beginning, but now I have the feeling that we're getting close to the core problem really quickly.

Everything is looking sound and reasonable, from a business point of view, except for a couple of steps whose meaning I still can't figure out. The domain has a lot of hard-to-understand regulatory constraints and bureaucratic steps, but these ones still look *really* awkward to my fresh eye.

The steps *are* awkward. It turns out that the internal software is imposing a couple of extra process steps in order to enforce consistency of the internal data structure, but this is unnaturally pushing complexity outside the software[2], forcing users to do extra activities outside the software.

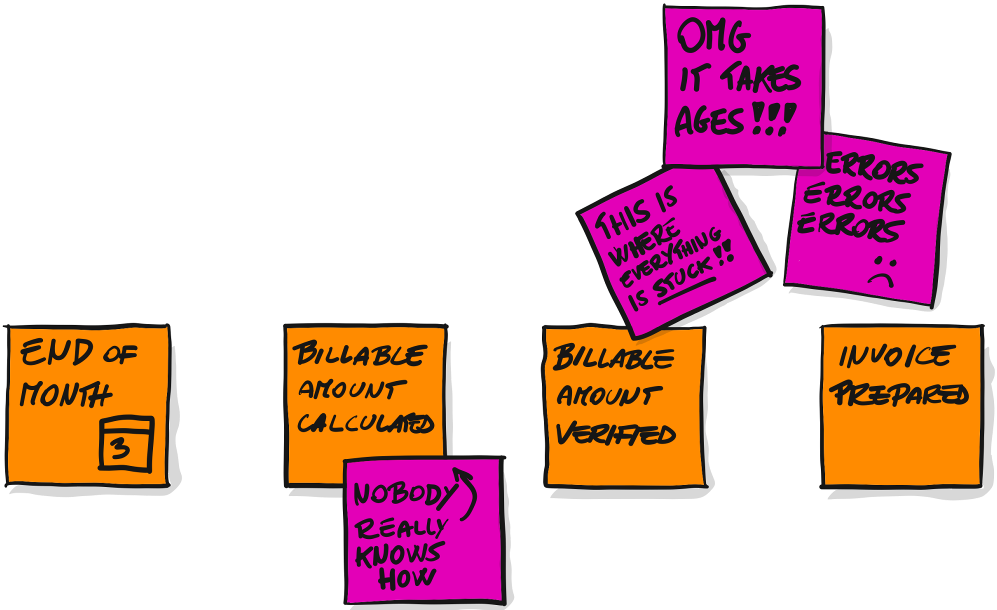
Once explored and visualized, the resulting real world process looks bumpy, and error prone. With a disappointing consequence: the process ended up *postponing customer acquisition*, not the smartest move in a competitive market.

Luckily, *we invited the right people*: decision makers are in the room. They propose a variation over the existing process on the fly, relaxing constraints for mandatory documentation. In fact, the currently applied requirements are unreasonably strict, so getting rid of their unintended consequences is an easy shot.

Time is getting short, so we ask if all the current blockers are currently displayed in the process (we added a couple more) and we asked participants to prioritize them. They almost unanimously point towards one. Easy.

---

[2] This is a pretty recurrent modeling anti-pattern, I named it *hypocrite modeling*. We'll talk about it in the modeling in the large section.



Looks like we have a clear winner. This is where we need to focus.

We have a mission, now.

Aftermaths

In the afternoon, the next action become obvious: since we highlighted the main blocker in the flow, there's nothing more important to do than *solving that problem* (or at least trying).

Finding a solution won't be as easy as spotting the problem. And the blocker is only the visible symptom, not the root cause. But *choosing the right problem to solve is a valuable result* and we achieved it *quickly*.

There's a feeling of urgency to prototype a solution. So, after taking pictures (in case someone tears the paper roll down), I start working on an alternative model for the selected *hot spot*, together with the software architect, based on two separated models instead of a single bloated one.

We need the prototype to show viability for an alternative solution, however we acknowledged that it won't be enough: now most of the impediments are political.

Even if we established a clear link between the major process blocker and a possible solution, putting it to work, is not only a technical issue: permissions needed to be granted, and roles and responsibilities in the organization around that step needed to be redesigned in order to overcome the current block.

Despite the initial assumption that software was the most critical part, it turned out that the solution was mostly *politics*. And when it comes to politics, transitions usually take more than expected, and they're never *linear* or *black-and-white*.

Kicking off a startup

Founders of a new cool startup are hiring a cool software development company and a cool UX agency to help them write their cool backbone software.

The business model is new for the market. In the founders' brain it's clear. They are very experienced in the financial domain. The development team had already been exposed to Domain-Driven Design; but developers, the ones supposed to make the thing real, have no previous domain knowledge - excluding some regrettable experience as customers - and the whole thing looks *huge*.

I am invited to join as a facilitator for an exploration workshop: the goal is to have the development team up to speed, learning as quickly as possible about the domain. Moreover, founders know about the business domain, but are unaware of the technical risks hidden at the implementation level.

Given the amount of mandatory compliance constraints in the domain, the

Minimum Viable Product[3] is going to be relatively big, compared to the ideal startup Eric Ries's style. Anyway we'll need to keep under control the natural tendency to inflate the MVP, since hitting the market as soon as possible is critical.

Day one

At 9 am, eight people are in the room: business, technical, and UX perspectives are represented.

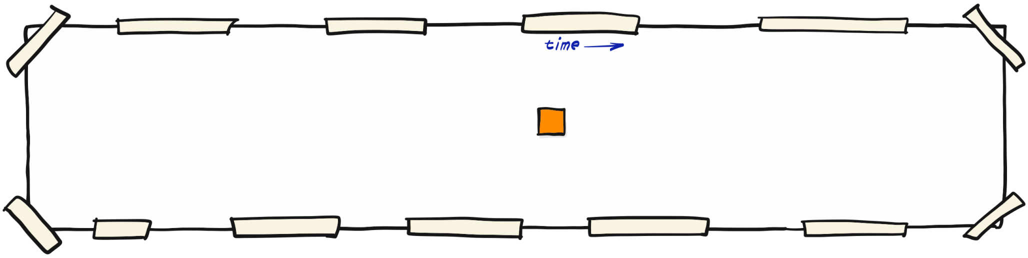
While everybody is just introducing each other and having coffee, I lay down my secret weapons: I unroll as much as I can of a plotter paper roll and stick it to the wall, then extract some hundreds of orange sticky notes from my bag and put them on the table, together with a ridiculous amount of black markers.

Some people already know about EventStorming, but the founders are totally new to the concept. However, they like the idea. The unusual setup already triggered curiosity, so I don't have to spend much time introducing the workshop. Instead, I immediately ask everybody to write the key *Domain Events* of the business process on orange sticky notes and place them according to a timeline on the paper roll.

The request sounds somewhat awkward, because we're skipping introductions and explanation. What we really need is an example: once the first Domain Event is in place, everybody is more confident about what a Domain Event can be.

---

[3] A *Minimum Viable Product* is one of the most intriguing and debated ideas presented in Eric Ries's book *The Lean Startup*, a *must-read* for every startupper in the digital world. In order to gather feedback from your potential customers as soon as possible, an MVP is the smallest possible product that can be shipped. The goal is to avoid wasting money building features, only to discover that no one would buy them.



Gotta start somewhere…

We quickly enter a state of *flow*: once the first Domain Event is placed on the timeline, it becomes fairly obvious what has to happen *before* and what happens *after*, so everybody starts contributing to the model.

Some orange stickies are duplicated: probably somebody had the same idea at the same moment, others look alike but with slightly different wordings. We keep them all, for the moment. We'll choose the perfect wording later, once we'll have some more information available.

I also remark that *"there is no right one"*. In fact, looking for the perfect wording will slow us down.

A few stickies do not comply with the Domain Event definition I provided, there's some sort of *phase* instead of a *verb at past tense*. But *phases can hide complexity* from our investigation, so I just turn the sticky note 45\Â° anticlockwise, to signal that something is wrong, without disrupting the discussion flow.

This is not a Domain Event, this is a process…

The modeling style feels awkward but powerful. We try to model processes before the detailed explanation from the domain expert.

There is no description of the boring, *easy-to-guess* steps of the business process. But every time the developers are off-target, then a meaningful conversation arises. We don't talk about *"the user needs to login in the system"*, (*boring*) instead we talk about places where there's more to learn.

It's more *discovery* than *translation*. This way, it feels like developers are really getting into the problem.

Conversations, conversations, conversations

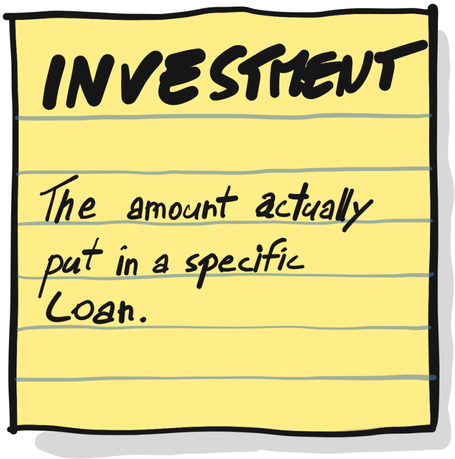
Sometimes the founders provide us with really cool insights about the unexpected complexity of the domain, and we listen like pupils in a classroom, taking notes, trying to formalize the just learned complexity in terms of Domain Events, adding more precise terms to the vocabulary.

When new terms arise, and the discussion shows that they have a very precise meaning in that *Context*, I start capturing key term definitions on a special sticky note and place them just below the normal flow.

It won't be a *Wikipedia-ready* definition: just the precise meaning of that term in that specific conversation.

Sometimes the knowledge is on the technical side: a developer takes the lead explaining the complexity that lies behind an apparently simple problem.

We could actually see the business and the technical parties getting wiser every round.


INVESTMENT
The amount actually put in a specific Loan.

When precision emerges from the words of a domain expert, better write it somewhere…

Open ended conversations are my favorites. In a couple of occasions the founder candidly says: *"I have no idea. We need to explore this scenario."* I loved that. Honest domain experts admitting they don't know something are a million times better than a *wanna-be-domain-expert* mocking up answers to stuff they have no clue about.

While people keep exploring different corners of the domain I start annotating all the hot spots in purple stickies. They may represent *problems*, *risks*, *areas that need further exploration*, *choice points* where we don't have enough information yet or portions of the system with *key requirements*.

Eventually, we start mentioning *external systems*: they can be external organizations or services, or online apps. I start representing with larger pink stickies the external systems our cool new software will have to deal with. This will happen in a more explicit fashion later on, but having some examples already available will make the following steps easier.

[FIXME: little image here]

As a background process, I start to label specific areas of the business flow[4].

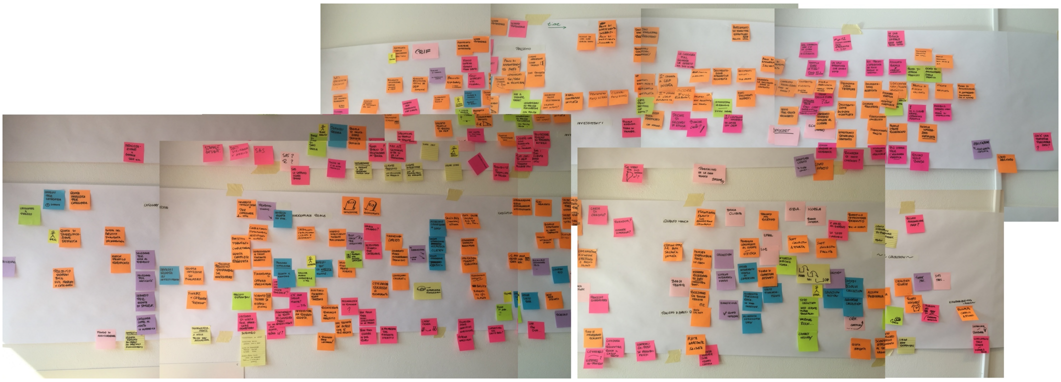
---

[4] I have a special tool for that, a removable label that can be moved around just like a sticky note. I'll talk about the tooling in the Tools section.

Something like *Customer Registration*, *Claims* or *Fraud Detection*. We're starting to see independent models that could be implemented independently, with different, relatively well-defined purposes.

I am not expecting precision and clean boundary definitions at this stage, just some emerging structure that we will improve later.

It turns out that there are already a few solutions available off-the-shelf, for given portions of the flow: no need to develop them in house, unless we need to do something distinctively different.



Too big to fit into one picture, but still manageable.

At the end of the day we filled up 18 meters of process modeling. A single wall isn't enough to contain the whole stuff, so we used the opposite wall too. We acknowledge that there were some more areas to be explored but the overall picture of the process looks solid.

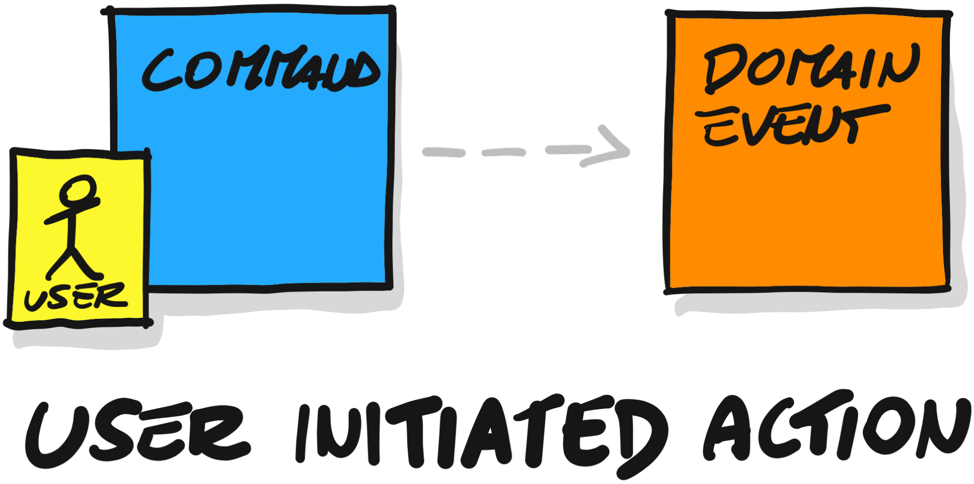
After contemplating our artifact, we head for dinner with the feeling of having accomplished a lot.

Day two

We start the session examining the model we left from the day before.

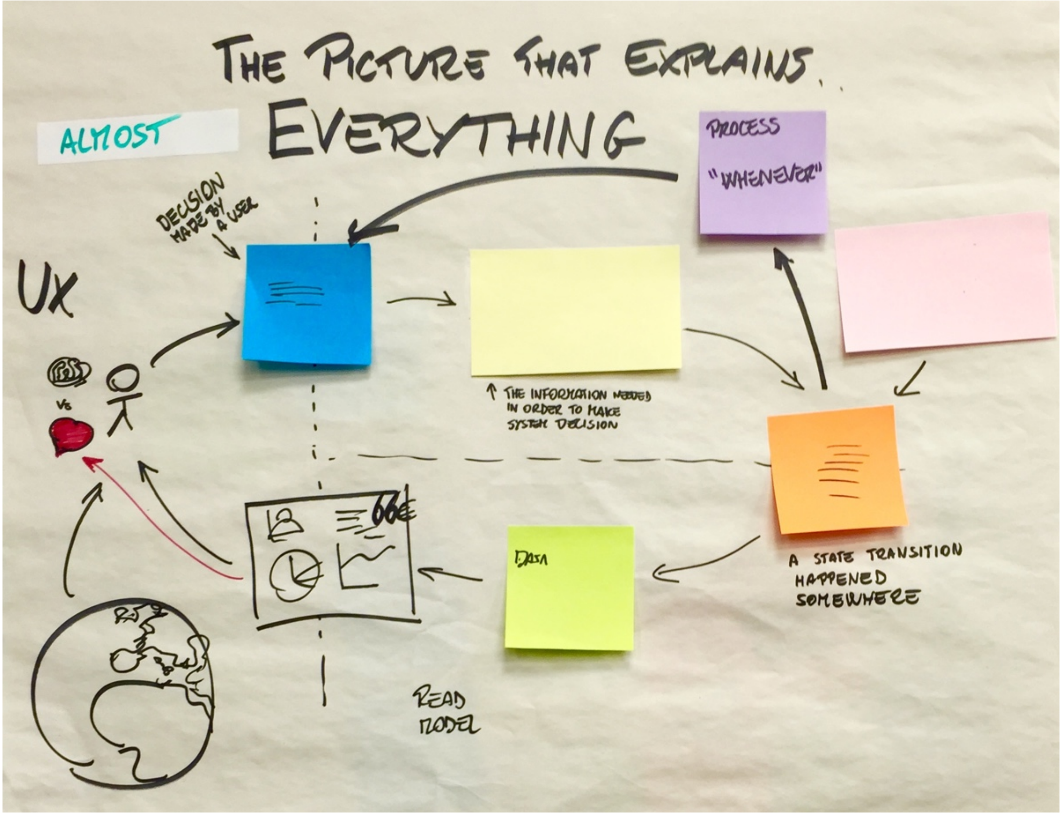
Some stickies now look naive compared to our new level of understanding. We rewrite a few with more precision and discarded some of the purple question marks that have been covered in yesterday's session.

It's time to get a deeper look into the mechanics of the core components of the system: we start introducing more rigor in our process by introducing *Commands* representing *user intentions/actions/decisions* (they are blue stickies where we write something like **Place Order** or **Send Invitation**), and *Actors* (little yellow stickies) for specific user categories.



Adding the little yellow sticky can make the difference, in shifting the focus towards user interaction

To provide a little more background, I draw *The picture that explains everything* (see chapter The picture that explains everything for more on that) on a flip chart and keep it as a reference.

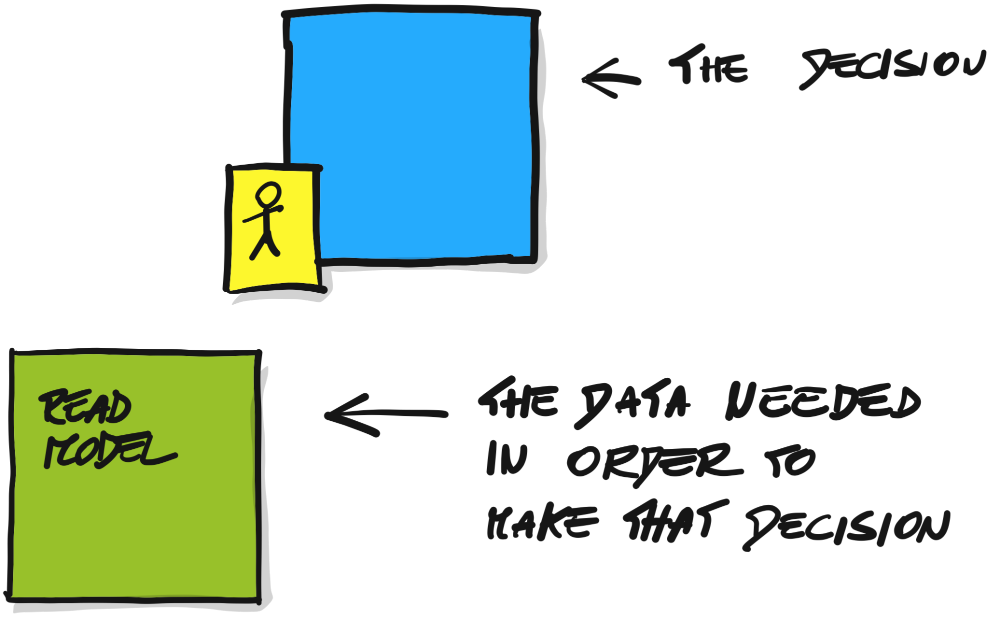

THE PICTURE THAT EXPLAINS...
ALMOST
EVERYTHING
PROCESS
"WHENEVER"
DECISION MADE BY A USER
UX
VS
THE INFORMATION NEEDED IN ORDER TO MAKE SYSTEM DECISION
66€
DATA
A STATE TRANSITION HAPPENED SOMEWHERE
READ MODEL

The incredibly ambitious "picture that explains everything"

Adding commands and specific actors triggers more discussions about the reasons why given users will perform given steps. Once more, some steps are trivial, while a few of them trigger deeper discussions like: *"Why should user X activate the service?"* which end up challenging the original assumptions.

A couple of interesting things happen around commands and actors. Commands are ultimately the result of some user *decision*, but thinking in terms of user decisions (*Can we make it easier? Can we make it better? Can we influence it?*) forces us to think in terms of the relevant data for a given decision step.

We capture this information in *Read Models* (green stickies in our colored dictionary), or even in little sketches if there's some key emerging requirement, like *"we need an image here"*, or *"price needs to be displayed in the center of the screen"*.



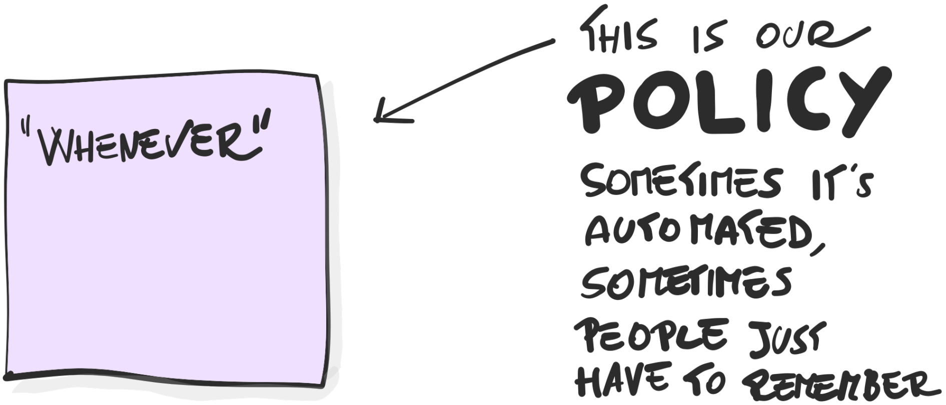
Read models are emerging as tools to support the decision making process happening in the user's brain

Interestingly, thinking about the users' motivation we realize that not every user in a given role thinks in the same way. Even if the decision might be the same, the reasons behind it will vary a lot.

What used to be a simple *user* or *actor* turns out to be a more sophisticated collection of *personas*. Since we're in Lean Startup mode, it is nice to see this concept emerge: we might eventually use it to sharpen our MVP by focusing only on few selected personas.

However, we don't care about the exact definition. The fact that *we're having this conversation* is way more important than the precision of the used notation.

The more we look into local mechanics, the more we find ourselves thinking in terms of *reactions* to specific Domain Events. It's easy to pick them up since they mostly start with the word 'whenever'. It's something like *"whenever the exposure passes the given threshold, we need to notify the risk manager"* or *“whenever a user logs in from an new device, we send him an SMS warning”*. Lilac is the color for this reactive logic that takes place after an event, and triggers commands somewhere else. ‘Policy’ is the name we end up using in this specific workshop.


"WHENEVER"
THIS IS OUR
POLICY
SOMETIMES IT'S AUTOMATED, SOMETIMES PEOPLE JUST HAVE TO REMEMBER

Our policy, regardless if it’s manual or automatic

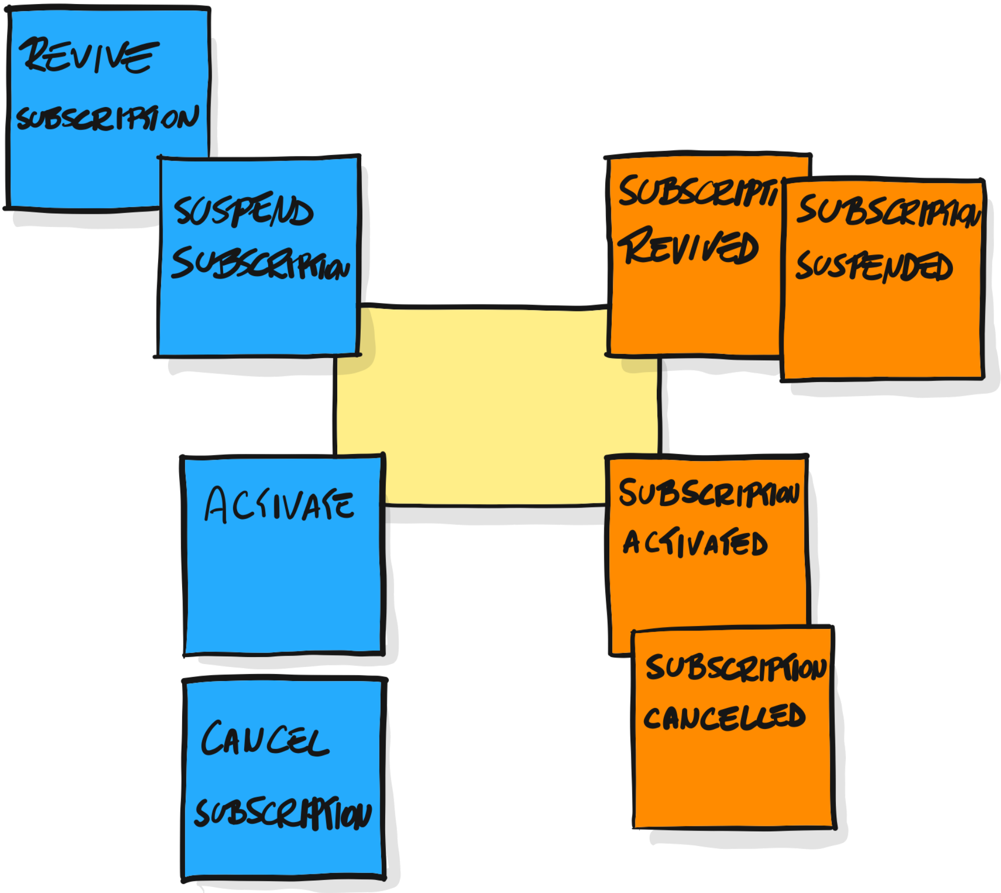
In a startup, it’s interesting to capture policies as early as possible, without making assumptions on their implementation. Even if some actions will have to be implemented by a system one day, it’s usually better to defer commitments (and the related expenses for licenses and fees) to the moment the customer base will be big enough to justify the investment.

A manual, possibly cumbersome, process today will eventually be turned into a software solution tomorrow. This way we can manage growth in absence of precise information, market feedback above all.

Developers are getting excited because we’re getting technical and an Event-Driven architecture is emerging from the mass of stickies, but business people aren’t excluded from the conversation, the reference notation is still visible on the flip chart acting as a visible legend and we’re mostly discussing the whys, or providing good insights on the complex areas.

Eventually, while founders move to a different office, we keep exploring the model with developers. They were eager to get into technicalities like cats smelling fish.

I agree, this is the time. Key *Aggregates* (in the traditional pale-yellow color) start to pop-up, in a responsibility-driven fashion. Given the bounded contexts have already been sketched, most aggregates look somewhat trivial.



A very simple aggregate, apparently

Interestingly, the exploration feels complete even if it's totally process-oriented. We had pretty much everything sorted out without using words like *database*, *table*, and *query*. Data comes in mostly in terms of mandatory requirements, or as a read model required to support a given user step. Only a few places still look like CRUDs but most of the business is described in terms of Domain flowing in the system.

And it feels so right.

At the end, it looks like we've covered pretty much everything. What looked like an unholy mess at the beginning, now looks like a reasonable flow. The team confidence is sky high. There are a couple of areas where experiments will be necessary (mostly because we need to understand system dynamics after hitting very large amount of users) but the overall feeling is that development would be on tracks, because everybody knows the reason why given pieces are there, and a lot of typical decision that will look stupid in retrospective won't be taken.

We are in *"the startup's twilight zone"*: assumptions should be challenged with experiments, but choosing the right assumptions to challenge and design the corresponding experiment is still largely an art.

Designing a new feature for a web app

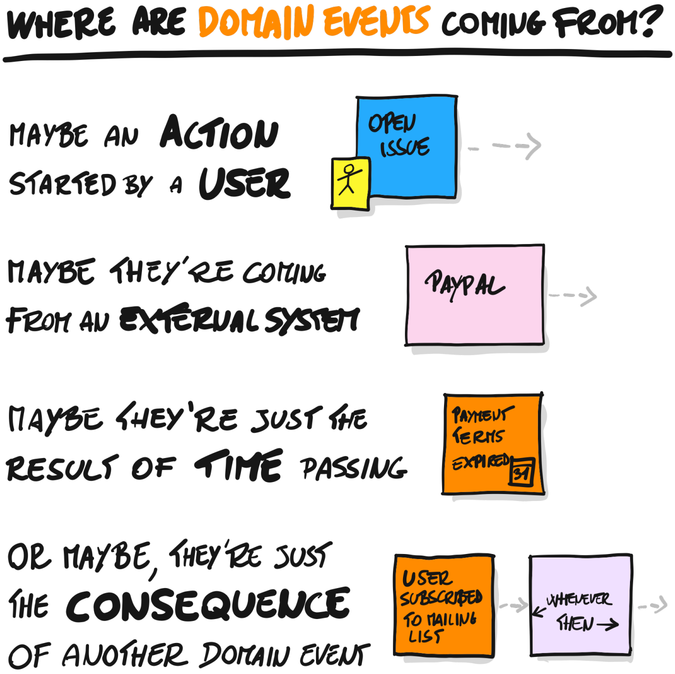
We just run a big picture workshop in the morning, but now it's time to narrow down scope and focus on the new requirements for our app.

We don't need every domain expert here: the scope of the new feature is relatively clear and involves only two of them.

[FIXME: probably a little later] With the development team we dig deeper into the semantics of the Domain Events.

It's funny to see how many times we're *rewriting* the sticky notes with different names, exploring variations or adding more precision to the picture. Even if we're throwing away many stickies, we still have a feeling of progress and increased solidity.

At this moment, we start getting deeper into the mechanics of our business process: every Domain Event should be the result of something. I sketch something similar to the picture below on the visible legend.


WHERE ARE DOMAIN EVENTS COMING FROM?
MAYBE AN ACTION STARTED BY A USER
OPEN ISSUE
MAYBE THEY'RE COMING FROM AN EXTERNAL SYSTEM
PAYPAL
MAYBE THEY'RE JUST THE RESULT OF TIME PASSING
PAYMENT TERMS EXPIRED
31
OR MAYBE, THEY'RE JUST THE CONSEQUENCE OF ANOTHER DOMAIN EVENT
USER SUBSCRIBED TO MAILING LIST
WHENEVER
THEN

Where are domain events coming from? Four different possible answers to the question.

- maybe a *Command* (a *blue* square sticky note) triggered by a given *User*;
- maybe the Domain Event has been triggered by some *External System* (normally a *pink* larger rectangular sticky note);
- maybe it's just the result of *time* passing, without any particular action involved (like `PaymentTermsExpired`);
- or maybe it's just the *consequence* of some other event: *whenever* one thing happen, *then* another another one will happen. Well, it's not always that obvious, so we set up a lilac sticky note for that.

Some developers are puzzled. A few commands look just like the rephrasing of the corresponding domain events. It's always funny to see how developers' brain reacts to boring and repetitive tasks. In fact that's not a problem: the whole thing is complicated, but this doesn't mean that every little brick has to be a mess. Some will be trivial.

However, in given portions of the flow, this is not happening: the relationship between an user action and a system reaction is more sophisticated than we initially thought, and can trigger different consequences.

Working with developers always brings an interesting consequence: they're wired up in thinking with alternative and complementary paths, looking for some form of *semantic symmetry*: so if we had a `ReserveSeat` command, they immediately look for a `CancelReservation` and start exploring the alternative path. This lead us to find more events and more *"what happens when ...?"* questions.

It's now time to introduce *Aggregates* one of the key concepts of Domain-Driven Design, by looking for local responsibilities that could trigger different responses to commands. We use plain old yellow stickies for that.

It feels natural to start grouping commands and events around the newly found aggregates, and this triggers even more discussion in the hunt for symmetry. Aggregates now are starting to look like little state machines.

Some people are puzzled because the moment we started to group around responsibilities, we broke the timeline. That's fine, the two groupings are orthogonal: we can't have both. The timeline was great for triggering big picture reasoning, but now *responsibility* is a better driver for robust system design.

The picture that explains everything is our companion again, and I refer to it in order to introduce *Read models* in green, to support user decisions and *processes* in lilac, that take care of the system reactive logic.

[FIXME: Finish!]

Quick EventStorming in Avanscoperta[5]

It's time to run a retrospective in our company. Training class sales haven't been as good as expected in the last weeks, and it makes sense to explore the problems that we surfaced along the way.

Since we advocate large modeling surfaces, we have plenty of space to model: every available wall of hour headquarter (a single 60 square meters room) is writable, so there's no need for the typical paper roll here. However, a large supply of orange stickies is mandatory.

There's not so many of us in the company. Actually, the company core is pretty small: only 4-5 people. However there's a significant amount of autonomy to take process decisions, and we experimented quite a few changes lately, so it makes sense to merge the different perceptions of the process into a consistent whole.

Everyone of us gets a marker. We start writing *Domain Events* (very short sentences like **Training Description Published** or **Ticket Sold**) on orange sticky notes, and place them along a timeline, from left to right.

There is no strict consequentiality: everyone writes about the topic they know best and find the corresponding place in the flow, eventually adjusting the position of surrounding stickies when things get too packed or too clumsy.

Along the way, we realize we keep mentioning External Systems (we have the pink larger stickies for that) and something that doesn't belong to the usual EventStorming toolkit: *communities*. They are meaningful enough in our context to justify a specific notation: we pick a color (light blue) that hasn't been assigned yet and add it to our incremental notation.

When the overall flow starts to make sense we start exploring issues and write them down as purple sticky notes and place them close the corresponding domain events. In our case, we had something like **Training Class Description Sucks!** and **Unclear Discount Policy**, or **Advertising Policies!!!**

---

[5] Avanscoperta is my small company, where I try to eat my own dog-food. www.avanscoperta.it

to highlight that we didn’t have a repeatable approach to advertising. Ideas come together and we end up with around 20 possibilities for improvement.

The outcome of our EventStorming session in Avanscoperta (one among many).

When exploring on such a wide scope it’s easy for issues to pop up randomly mixing the very important things with the trivial ones, so - just like we would do in a retrospective - we’d prioritize issues, with the impact in mind: _“what is the issue that’s going to have the biggest impact when solved?”_Â We decide to focus on the appeal of the product we’re selling. The decision to buy a ticket for a training class - which we’ll represent as a blue sticky note - is a user decision which will involve both logical and emotional thinking.

On the logical side, we have a little problem with the data displayed: though current price is currently displayed, our naming policy for discount is somewhat confusing for the users: two prices are shown and some user wonder whether they lead to different service classes. Maybe not a big blocker, but the buying experience looks clumsy.

The big blocker stands right in our faces: the purple sticky note saying **Training Class Description Sucks!** stands right in front of us, telling us that the perceived value of what we’re selling is not enough to trigger the purchase.

A quick look to the training class description of our best sellers, compared to weaker ones confirms the hypothesis: despite good teachers and cool topics, the class as shown on the website is not triggering any strong call to action.

Now we know that we should put a lot more effort on crafting the content of the training class: it has to speak to the rational mind, but it has to be catchy too for the emotional part of the brain (or ‘the heart’ if you are feeling more romantic than scientific).

In the end, my brain is torn apart. As an entrepreneur, I am quite happy I've identified problem number one on the list and now I have an action plan towards a possible solution. As a developer, I am slightly disappointed, since the solution doesn't involve writing any code.

I trust the next bottleneck to be better.

You've now got a little taste of what an EventStorming session might look like. There's something different in every single approach, and many choices are largely context dependent. Some similarities are visible though:

- an *unlimited modeling surface*;
- a virtually *unlimited supply of markers and sticky notes* in different formats;
- a *collaborative modeling* attitude involving all key roles and stakeholders;
- *no upfront limitation of the scope* under investigation;
- focus on *domain events* along a *timeline*;
- continuous effort to maximize *engagement* of participants;
- *low-fidelity incremental notation*;
- *continuous refinement* of the model, or *progressive precision*.

Possible formats

The discipline[6] is relatively new (my first experiment dates 2013), but there's already some distinct established format. I like to thing about it like Pizzas: there's a common base, but different toppings.

Here's my menu.

---

[6] I had shivers down my spine when I realized I associated the word 'discipline' with the apparent chaos of EventStorming.

- **Big Picture EventStorming**: the one to use to kick off a project, with every stakeholders involved.
- **Design Level EventStorming**: digs into possible implementation, often DDD-CQRS/ES[7] style.
- **Value-Driven EventStorming**: A quick way to get into value-stream mapping, using storytelling like a possible platform.
- **UX-Driven EventStorming**: similar to the one above, focusing on the User/Customer Journey in the quest for usability and flawless execution.
- **EventStorming as a Retrospective**: using domain events to baseline a flow and expand the scope in order to look for improvement opportunities.
- **EventStorming as a Learning Tool**: perfect to maximize learning for new hires.

In the next chapters we'll start from the Big Picture approach, the one that opens the way for further explorations.

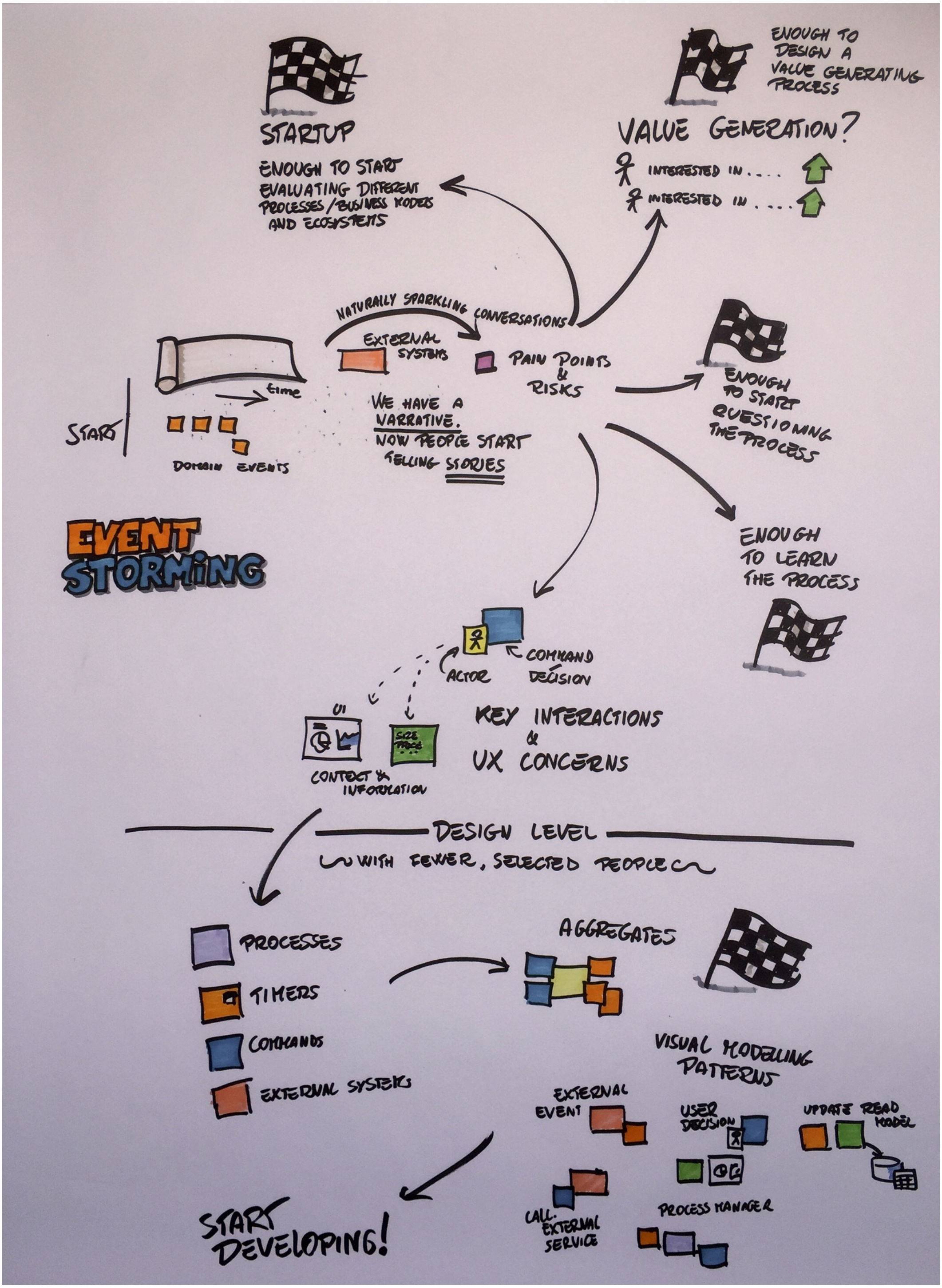
* * *

Chapter Goals:

- a narrative about what happens in a good workshop
- a glimpse of what can be achieved
- a glimpse of the underlying mindset
- an anchoring about: "I want to get there!"

---

[7] Domain-Driven Design, Command-Query Responsibility Segregation and Event Sourcing. We'll talk about this later.



An overview of some possible different paths

A deep dive into problem space

“The only truth about human condition is that everybody lies, the only variable is about what.”

Dr. Gregory House[8]

This section digs into the nature of the problems EventStorming is addressing. They’re big, and deeply entrenched. Despite all my efforts to make this section lightweight, it is big.

Don’t look at me! It wasn’t me creating the problem. It was human nature, decades of wrong assumptions about software development, and so on. I just ***need to expose the problem*** *in all its scary magnificence.*

In Chapter 2: ***A closer look into problem space*** *we’ll look at the typical problems that we might face when running a Big Picture EventStorming. We’ll (re)discover some classical organizational dysfunctions, and maybe highlight something unexpected.*

In Chapter 3: ***Pretending to solve the problem writing software*** *we’ll dig into the typical fallacies of software engineering and why some common approaches are unfit to attack the problems we’re supposed to solve.*

The two chapters are complementary, and should be read one after another. However, If you’re eager to get into the mechanics of your first workshop, you might skip chapters 2 and 3, if you promise to come back.

You won’t regret it.

---

[8] https://www.youtube.com/watch?v=CZR2k5c_198

2. A closer look to the problem space - 80%

In every workplace, there are two categories of people: the ones who understand complexity and the ones who don't.

Complexity in a nutshell, or less

Complexity is a daunting topic. It scares people by definition - *"OMG this is SO complex!* - And the typical entry point on the matter is usually a conference speaker remarking the difference between the words *'complex'* and *'complicated'* that sound alike to the average human being, and mean almost the same thing, but are *extremely* different once we enter the realm of complexity[1].

[FIXME: something is missing here]

However, when it comes to organizations where people are working together towards a common goal, we can place a safe bet that we are in some form of a Complex Adaptive System. You may now want to go on Wikipedia for a more formal definition[2], but I am all for simpler, more actionable ones.

So, my unofficial definition of a Complex Adaptive System is

a network of interdependent agents influencing each others

---

[1] not to mention the inevitable question about *"how do you pronounce Cynefin?"*

[2] https://en.wikipedia.org/wiki/Complex_adaptive_system

Your company, your community, your family, they all can be seen as Complex Adaptive Systems.

What does it mean in practice? Well …the best rule of thumb I know is that

a complex adaptive system won't behave as expected.

Here's a couple of stories to get the taste of it.

In TXZ (a not-so-fictitious company) the boss "asked" employees to arrive in the office strictly before 9:00 am, in order to improve productivity. Employees complied. Despite working often till late hours, people started to wake up earlier and faced the morning traffic in order to show up at the office on time, and swipe the badge. However, employees adjusted the habits and started having a little breakfast and coffee, after they checked in. Having breakfast together with colleagues triggered some extra gossip. Day after day, the breakfast turned into a social activity, and even the ones that already had breakfast at home, joined just for a cup of coffee. Real work wasn't starting before 9:30. The actual working hours actually shrunk.

* * *

Kaboom! (another not-so-fictitious organization) once adopted a policy of incentives based on the bug resolution time, aiming to improve the quality of their software. Once employees got familiar with the new system, they started tweaking the policies for opening bugs: many trivial bugs were entered in the issue tracker on 'easy days', and solved shortly after. Metrics (and the related bonuses) improved, but the software quality didn't improve. In fact, relevant hard-to-solve issues reported by real customers were put in low-priority behind the scenes.

Can you spot the problem? Yep, it's *human beings and their annoying ability to make individual decisions,*[3] adapting to the surrounding context. Or *free will* if you like it.

More interestingly, it turns out that orders, incentives and superimposed policies are often really unsophisticated ways to change the people's behavior at a system level. Individuals will make decisions based on a broader context, that includes their own personal well being, their peers behavior, their ethics, and so on.

---

[3] Maybe I have been influenced a little too much by *The Matrix* saga.

Some may label the situations I described above as *cheating*[4], but if the system is fooling you, if people are not behaving the way they're expected, chances are you're dealing with a Complex Adaptive System and using the wrong strategy.

More interestingly, individual choices often result in *emerging behaviors*, like the 'morning gossip time', or the 'easy bug conspiracy' that could reshape the whole system in unpredictable ways.

Complex systems are non deterministic

Complex systems are not easily predictable. It's not bad luck: *it's their intrinsic property*.

When something unexpected happens, people will find an explanation in terms of cause and effect *after the fact*, but it won't look that obvious before the thing happens. You might be aware that you're dealing with a system, but the actual constraints of the system will be understood only in retrospective.

It actually took me a while to get familiar with this notion. Having a background in computer programming I tend to have a deterministic mindset. A computer software MUST be predictable, even in the few cases where the outcomes are fooling us.

When a deterministic system behaves in an unexpected way, it means that there is some hidden detail that we're not considering. Once we find out the missing piece, we might be able to explain and eventually reproduce the problem.

---

[4] To be honest, I find the whole idea of cheating intriguing. It is the result of two conflicting forces: contextual need and the personal reward of seeing ourselves like good persons. The term has a strong negative connotation, but in practice is more context dependent than we'd like to admit. In sports, cheating is intrinsically evil while stopping at a red traffic light in a desert street will make you feel stupid. In war, a fake document could save human lives; would still you call it cheating?

But systems made of people. Well …they're another matter. One way or another, they'll always surprise you. People learn, and usually you can't fool them twice in the same way.

Retrospective explanation

After the fact, somebody will always provide a good explanation, with the annoying property of sounding *so obvious!*

Well, at least we learned something.

Can we really learn a lesson?

Unfortunately, even retrospective analysis is not guaranteed to find the real reason things happened. Deriving lessons from past experiences can be a slippery slope: humans have the innate need to look for *simple narratives* to explain complex situations, chasing scapegoats (it's so reassuring to put the blame on someone else instead of on a system we belong to), bad luck or whatever is functional to preserve our own personal status quo.[5]

Let me be more explicit:

- In complex systems a cause-effect relationship will be evident only in retrospective[6]. This is not bad luck, it's *by definition*.
- *Evident* doesn't mean *understood*. Human brain is wired to look for simple cause-and-effect relationships even in situations where they don't apply. A *simple*, *plausible narrative* will often be the preferred choice to explain unexpected outcomes, over a more sophisticated one[7].
- In a corporate environment, or in politics, the typical simple and plausible narrative would be to blame scapegoats, and subsequently fire (or promote) them. If fingers starts pointing towards the designated ones,

---

[5] This is a major combo! I managed to reference 3 groundbreaking books in one single sentence. I am referring to Daniel Kahneman's *Thinking fast and slow*, to Nassim Nicholas Taleb's *The black swan* and to Matthew Syed's *Black Box Thinking*. Each one is highly recommended.

[6] As a bonus, you'll always get a free round of *'I told you it was going to end this way'* by one of your colleagues.

[7] Even this one is incredibly well explained in Daniel Kahneman's Thinking Fast and Slow

then *'bad luck'* or *'exceptional circumstances'* will be the words most likely to come out of their mouth.

- Putting the blame on the system isn't sexy. *System is everyone*, and doesn't provide easy narratives. Moreover, after a critical problem emerges, people badly *need* to declare it closed. The anxiety and the guilty feeling related to a pile of unresolved problems is often too much to bear. We need to sleep, after all.

Unfortunately, the bare truth *'we are pretending to lead a system whose behavior is still a mystery to us'* isn't a politically viable statement.

[FIXME: glue]

In complex systems *context* becomes the driving force: strategies and solutions are not safely repeatable; local contextual information will be making the difference.

The need for proven solutions

Conservative organizations are usually reluctant to fully embrace the consequences of complexity. Risk adverse culture manifest itself in the pursue for *proven solutions*, rather than adventurous experimental ones.

Unfortunately, in complex domains, there is no such thing as a "proven solution". Hiring the best coach with the longest winning strike will not guarantee that your team will win the championship next year.

In practice, being dogmatically risk-adverse turns out to be a very risky strategy in a complex domain.

The role of the expert

In places whose complex nature is misunderstood, the "Expert" becomes a controversial role. The common perception of the expert is highly influenced from complicated domains: a technician will tell us why our car is making that strange noise, why our ADSL isn't working and so on. We expect precise diagnoses and solutions. Physicians are often expected to provide the same level of certainty[8] even if they cannot be 100% sure of the diagnosis.

When entering the realm of complexity, experts have an expectations problem. People are in search of exact answers, and simple recipes. But the latter are the territory of demagogues, not problem solvers, with the notable exception of the few ones that managed to distill a well-targeted, sophisticated message so well that it stick with everyone[9].

Often, power play gets into the game: *"I need an answer right now!"* (with a big capital 'I') is a recurring meme when management is uncomfortable with high level of uncertainty. Apparently, a wrong answer *right now* sound more appealing than *no good answers yet*.

The bare summary is that in complex environments experts don't have all the answers. They'll have experience, analogies, ideas, and a network of peers to provide advices. But this won't guarantee they'll take the right decision.

Strategies for decision making

If relying on proven best practices isn't a viable strategy, the burden shifts on the decision making capabilities of the organization. There will be more decision to make, and they'll be more critical.

Since the outcome won't be guaranteed, many decisions will have to be treated like *experiments*. Scary as it sounds, this is the soundest strategy in a complex environment, assumed that you're putting some salt in designing the experiments, and in learning as much as you can from them.

But decisions do not happen for free. In fact, in most organizations, critical decisions are the most energy demanding activities. And here is where organizations expose their most annoying dysfunctions.

---

[8] But without the possibility of running an autopsy in order to have more precise data.

[9] You may think Gandhi or Martin Luther King, as notable examples. But if you're thinking business, the best example I can pick is the transformation that took place in Alcoa starting from the 'Safety First' principle. An interesting summary can be found in The Power of Habits by Charles Duhigg.

If only a few enlightened people are entitled to play with the system, then decision making power becomes an even scarcer resource. Some people might start to cut corners under the hood, in order to get something done, while others become masters in the bureaucracy needed in order to follow the rules.

[FIXME: glue]

Complexity plays also another subtle twist that challenges traditional organizations: where repeatable strategies aren't available, the need to make critical decisions on the spot becomes critical. But most of the key information needed to make sound decisions is now coming straight from the context, and it is usually closer to the people on the field than to the management layer(s).

In a layered hierarchical organization, decisions from above don't match very well with complexity. A decision from the management is at risk to arrive too late, after the information climbed up the layers in the hierarchy and to be flawed due to information lost along the way.

[FIXME: pyramid image]

In complex environments, the ability for the ones with the right information to take the right decision becomes crucial. *Self organization* is the hyped term describing the ability of a group of people to respond quickly and collaboratively to a problem.

Impediments to self organization

Unfortunately, self organization doesn't happen just because somebody found the time to read an article on the web.

Telling people to self-organize without creating the needed environment isn't enough to trigger the magic. We're still in a Complex Adaptive System, after all, and people will adapt to emerging situation in ways that we can't anticipate.

Like many good practices in complex systems, self organization is not guaranteed to happen. It *can* maybe happen after we create the right environment and remove road blockers[10] ahead.

The list of blockers can be long. A culture of blame and fear is the arch-enemy of self organization. People will have ideas, but if they're punished rather than praised, they'll learn to keep their mouth shut, at least in public, and to follow orders rather than propose solutions, or make experiments.

But even if *'every organization is special in its own way'* their uniqueness is often overrated. As systems, they expose common traits. Some situations are in fact *predictable* to a given extent. In practice, there are a few recurring blockers which are in the way of many good initiatives.

Blocker Nº1 is *the lack of understanding of the system as a whole*.

Understanding the system

University students sharing the same apartment will have to find ways to self organize around basic duties like cleaning, grocery shopping and cooking. It may take some time, but one day they'll be in the same room, have a little remarkable showdown and finally discuss different arrangements. It's easy to come up with an agreement in such a scenario: feedback loops are short and there aren't many external sources to blame[11].

In larger systems, self-organization may be harder to achieve: feedback loops will take a significant amount of time before displaying their effects; consequences of given actions will only be visible only somewhere else in the system (so it's not our problem anymore). In larger corporations, departments will be perceived as separate entities, or even be outsourced adding more opportunities for scapegoating.

We have a problem. -> The problem is coming from an external source. -> There's nothing I can do about it. -> Problem solved! -> I'll keep doing business as usual.

Well, clearly the problem *isn't* solved. But I suspect this reasoning is familiar to many readers.

---

[10] Road Blockers have this annoying property: they are almost *guaranteed to succeed*. To state the asymmetry in another way: we can guarantee failure, but we cannot guarantee success. As I said, …annoying.

[11] *"The pigeons stole the toilet paper"* is definitely not a good strategy for self organization.

However here are my main blockers to self organization.

People won't self organize around a system they don't understand

Without understanding of the system as a whole is very hard for people to self-organize.

People won't self organize around a system they can't influence

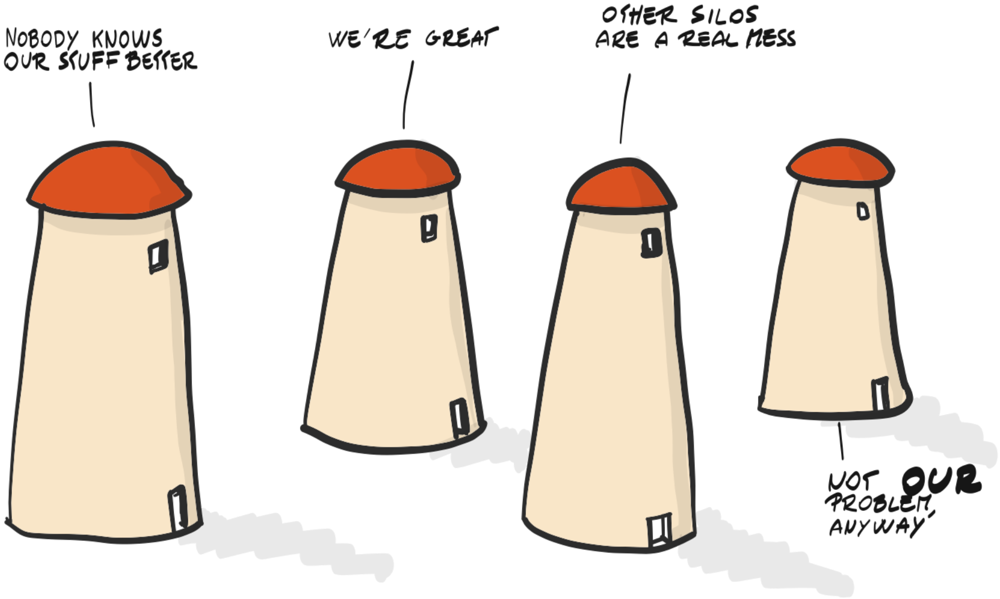
[FIXME: complete] Readability and consequences.

Organization silos

Organization silos are a good example of complicated thinking applied to a complex domain. If organizations were like *machines* then partitioning them into well defined units would look like a reasonable choices. But pistons don't go on strike, tires don't lose motivation and plan to go working for another car, and the gearbox does not pass its time complaining about the driver's style.

However, most organization end up being structured into some form of silos. Your organization doesn't have to be really large, to expose the symptoms. Even in very small ones, people invariably tend to do two things:

1. focus on *their own area of responsibility* (apparently a good idea);
2. progressively ignore what happens in other departments.


NOBODY KNOWS OUR STUFF BETTER
WE'RE GREAT
OTHER SILOS ARE A REAL MESS
NOT OUR PROBLEM, ANYWAY

Everybody is a master in his own silo

Traditionally, silos are supposed to be bad. They slowly undermine the organization ability to evolve and quickly respond to challenges, but from an evolutionary point of view they are the winning breed. So there must be something special in silos that made them so successful.

In fact, silos do excel in one thing: *they minimize the amount of learning needed for newcomers*. New hires don't need to learn the whole thing in order to feel productive, just the portion of knowledge required to get their job done. Specialization is both a byproduct of silos and an enabler for more future silos, *oooops!*

The reversal is more self-explanatory:

Silos maximize overall ignorance within an organization

…which might be a good thing if you're into something illegal or ethically debatable, but for most enterprise challenged by market and competitors, this will backlash sooner or later.

However, three things are really relevant here.

1. The forces driving every organization into silos are *strong*. They aren't evil: there is no conspiracy to turn organization into silos. There are people trying to do their work and applying plausible local solutions to local problems, while progressively losing sight of complexity growing where they stopped watching.
2. Those forces *will always be present*. Without an explicit balancing action, silos will inevitably emerge. It's our responsibility to face problem: doing nothing to mitigate the it *is* part of the problem.
3. Slowly, the problem will become *too hard to be solved*, or too expensive. Starting a new organization, or just *quitting*, eventually turns into a valid alternative to dedicating energies into a slow transformation of the current status quo.

Every organization exposes some degree of silo behavior and this is making the organization sub-optimal, less efficient or - in the worst cases - doomed to failure. The really nasty trait if silos is their asymmetry: they're easy to establish, and really hard to remove.

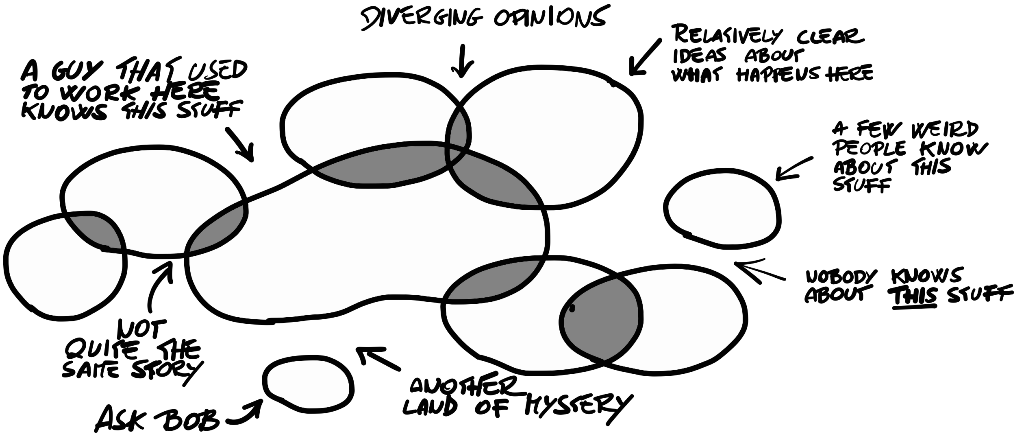
From our point of view, *this is actually a cool opportunity*: there are common problems emerging in a recurring fashion, and we may have a new powerful tool, targeted exactly where it matters most.

Uneven distribution of expertise

However, when we try to address problems in such a complex space, another property of silos gets in the way: they do enforce an *uneven distribution of expertise*. When multiple expertises are necessary, this will constrain the way we can gather the necessary knowledge in order to solve complex problems.

Complexity pulls for context aware learning and decision making, but silos partition awareness and make it incredibly costly to put the pieces of the puzzle together in order to find solutions to non trivial problems.

Usually we'll have to deal with different knowledges and expertises, and the information we'll achieve can only be locally consistent: nobody holds the whole truth.


DIVERGING OPINIONS
RELATIVELY CLEAR IDEAS ABOUT WHAT HAPPENS HERE
A GUY THAT USED TO WORK HERE KNOWS THIS STUFF
A FEW WEIRD PEOPLE KNOW ABOUT THIS STUFF
NOBODY KNOWS ABOUT THIS STUFF
NOT QUITE THE SAME STORY
ASK BOB
ANOTHER LAND OF MYSTERY

Overlapping and conflicting expertises, and ignorances too.

[FIXME: the picture should use clouds, not bubbles]

To be honest, the real situation is even worse. Even local consistency of knowledge is not guaranteed. Employees come and go. They learned the same discipline in different schools and previous workplaces. They can *disagree*, sometimes violently, or they may agree on the problem essence but use a different terminology to describe it. They keep mixing problems and solutions, or describe reality in terms of the existing software in use. It's a real mess.

Fine. Let's stop pretending that it is not.

Now, the simple idea of gathering requirements for a software project into this mess looks daunting. It's actually even *worse* than this, we'll talk about it in *Requirements Gathering is broken*, in a few pages.

Making money out of this mess

Organizations should have a *purpose*. Maybe to provide a service to customers or citizens, maybe to make a lot of money, or, like in James Bond's movies, to rule the world and feed white furry cats.

Every business will require a combination different expertises and skills in order to succeed. Consider a small on-line shop, like Etsy[12], for selling hand made items. Even a minimal business like this one will require multiple expertises.

- **Design**: the shop owner needs to think about what to produce.
- **Supply**: wool and knitting tools need to be purchased, if you're looking to produce something unique, finding special raw material becomes critical.
- **Production**: somebody's got to craft the handmade item. And has to be expert enough to craft something beautiful.
- **Content**: items needs to be described and displayed. A good picture and a bit of storytelling can make a lot of difference on the web[13].
- **Delivery**: different service providers will do the delivery job, however somebody will have to learn how to interact with them.
- **Packaging**: finding an elegant way to protect your goods could be tricky.
- **Claims**: you'll have to become an expert as soon as you get the first one.
- **Bookkeeping**: tracking costs and revenues.
- **Taxes and regulations**: you thought it was easy, then, for some reason, you discover that pink items have a different taxation in a nation you didn't know it existed.
- **Publishing**: the platform can hide a lot complexity to the users, but still someone's got to learn how to use the platform.
- **SEO**: one can't really think about selling online without getting addicted to traffic stats, and how to affect them, choosing the right keywords and so on.

---

[12] http://www.etsy.com

[13] AirBnB understood this principle so well that it started offering professional photo shooting as a service to their hosts, in order to improve the appeal of their rooms and apartments.

- **Community Building**: oh yes, it's *conversations* before sales. And communities need to be *engaged* too.

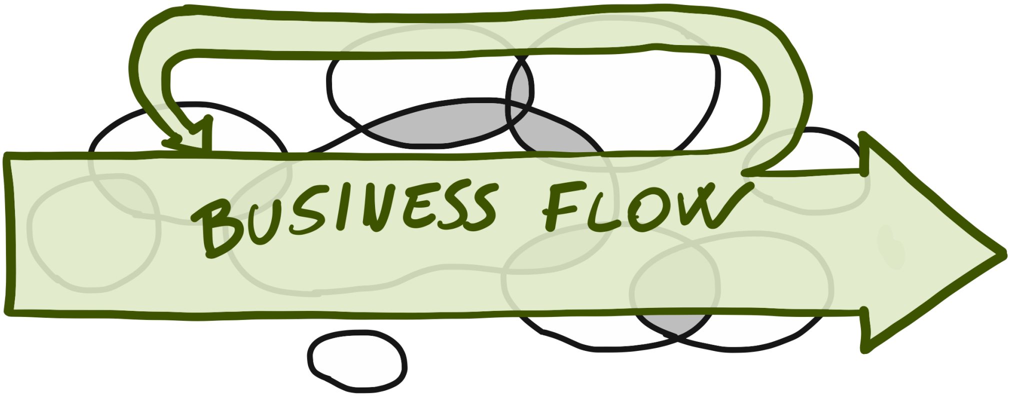
Hmmm… Not as easy as it looked. Now, think about a *larger* company. Hmmm…

However, the areas of knowledge one has to master in order to run a business are many, each one with its own specialties. Not everyone is required to be a great expert: one can be really good in something and just good enough in some other non critical.

They're not silos in our little Etsy shop, but we can guess why silos emerge: as the company grows, new hires will allow someone else to *ignore* certain areas. Nobody wants to know the whole thing.

Business flows are crossing the silos

Now, the business flow in enterprises isn't limited to a single area. In fact it normally spans across several areas of expertise



The business flow crosses silos and expertise boundaries.

The obvious evidence is that one single source of knowledge won't be enough to solve problems that are spanning across multiple areas.

Organizations

Organizations aren't perfect machines. Regardless of how much hype you put on the healing effects of free market, they aren't the amazing result of some Darwinian evolutionary force, like *survival of the fittest*. Well …they are, but only to a given extent.

Organizations don't have to be perfect. If they're on the market, they just have to beat competition, or find a niche that guarantees survival. It's not uncommon to find organizations which are dysfunctional even in highly competitive markets, especially if their competitors share the same cultural background.

Moreover, it's relatively rare to find organizations that grow according to a specific *design*. More often than not, organization react and make sense to the market drivers (including the job market) that shape them.

However, there's a

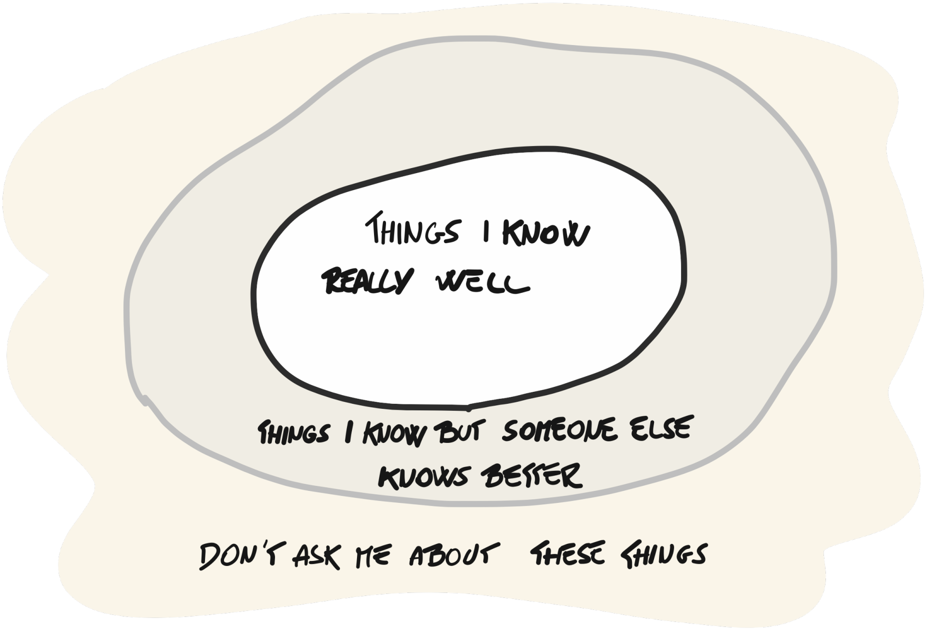
[FIXME: or delete it. :-( ]

Individual areas of expertise

Things don't get any better when we zoom into individual areas of expertise. What looked like bubbles in the previous pictures are in fact more blurred than we'd like.

- What experts *know* and what *they think they know* aren't necessarily the same thing. When asked about things they think they know, they may genuinely give you an incorrect answer. Put in another way: people will always give a *description of their current level of understanding of the problem domain*. Experience and good faith aren't enough to guarantee adherence to reality of this description.
- When experts are asked something beyond their area of expertise, they may give you the answer (which might be a *problem solved* for you, if you are investigating) [FIXME: maybe move in requirements gathering] but not necessarily mention that you should ask someone else in order to get more precision.

- This is particularly true when dealing with customers or with users which aren't inside the organization. Experts will tell with a good degree of precision why *they* are doing given things, but when they talk about users and customers they will basically offer data-backed speculation.


THINGS I KNOW
REALLY WELL
THINGS I KNOW BUT SOMEONE ELSE
KNOWS BETTER
DON'T ASK ME ABOUT THESE THINGS

There are things I know, and things I know very well.

Even in areas we know well, there can be inconsistencies. We don't always call things in the same way. I still remember an old teaching from my Italian language professor, that incited me to *use synonyms whenever possible* in order to avoid boring repetitions. Imagine the effect of this approach on a technical specification document.

Personal stories can mix in as well. A person coming from a different company, might have a different jargon and be in the process of mutating it into something different. In general we tend do consider language as a consistent tool to express our knowledge. It's not.

Moreover, people tend to confuse things. They mix facts and expectations: sentences like *"payments should always arrive on time"* might be incredibly misleading on the fact that _"payments may in fact arrive at a random moment in time, from the desired customer, hopefully with the expected amount."[14] and that we have only little control over that.

[FIXME: move it in the requirements gathering portion] The net result is that we can't blindly rely on what people tell us. We can't assume expert knowledge to be consistent, but - as learners - we don't

Hierarchy

The presence of a hierarchy is strongly correlated with organization silos. Hierarchy is a *resilient* organizational pattern, and for many people in different organizations, it's the one and only way to structure an organization.

Discussing alternative forms of organization goes beyond the scope of this book. I recommend to give a look to Dave Gray's excellent The Connected Company for an exploration of different organization structures, and the principles that guide them.

Just like silos, we might have to accept the presence of hierarchy as a *fact of life*. But we must take into account a few common consequences.

- Hierarchies *reinforce* silos. People get usually hired, trained and promoted inside a silo, and their duties and reward are normally within a silo boundary.
- Hierarchies partition responsibilities. But an organization can be way more complex than just *the sum of its departments*.
- Problems within silo boundaries are addressed quicker than problems spanning multiple silos, or in the *no man's land*. In fact, problems

---

[14] True story. A customer paid an invoice more than the specified amount, in order to pay back lunch to a colleague of mine. Imagine the accountant's eyebrows.

which aren't contained within a clear area of responsibility will tend to chronicize.

The clash between hierarchy and business flow

A couple of years ago, I had the chance to see Niels Pflaeging in action, in a *punch-in-the-face* session at Stoos Stampede, in Amsterdam.

He described how in every organization there are three main *networks* connecting people.

- The ***hierarchy*** as a top-down static structure, usually described by an organization-chart.
- The ***informal structure***, the affinity- and competence-based network connecting the people who you'd ask for help, on the many different topics.
- The ***value creation*** network, involving the people we have to collaborate with in order to deliver a service.

The bottom line was: *"Only the second and the third are creating value, the hierarchy is not."* Ouch[15].

The shape of the problem

Let's see the statements above from the other way round: there is a specific category of problems, which are usually long lasting in the organization. Sometimes they've been around for so long they're not perceived like problems any more: they're *just part of the landscape*.

[FIXME: this is a good place for a recap image]

This category of problems is

---

[15] A very good and short intro of Niels Pflaeging's point of view is on this post: https://www.linkedin.com/pulse/stop-crap-steps-towards-robust-theory-org-power-niels-pflaeging?trk=hp-feed-article-title-comment

- **hard to *solve*** because it requires an agreement by many different stakeholders,
- **hard to *discuss***, because it involves many different actors, usually trying to escape blame by doing things right in their own silo, and the conversation becomes an endless loop of *"you, know the real problem is [someone else's problem]"*;
- **hard to *visualize***, because it involves many interdependent aspects in different areas of expertise.

The good news

EventStorming allows learning to happen across the silo boundaries. It's not an initiative to transform the organization, but a too to understand what's going on, at a wider scale. We'll see how asking stakeholders to model a complex business flow together will expose many of the conflicts and contradictions happening at the boundaries between silos.

It will also help to look for system-optimal solutions, which is hard, because *the system is really hard to see*, instead of local sub-optimal solution and a lot of useless politics to justify them.

The bad news

The deeper level of company self-awareness is probably the most valuable outcome of a Big Picture EventStorming workshop. However this is also one of the worst selling point ever.

"The workshop will help you understand your company better"

Are you insinuating that *I don't know the company I've been working for all these years*? Even if you're right, and you'll deliver some precious insights, just do yourself a favor and don't use this as a selling point. Chances are this is a recipe for *political suicide* within your company.

Chapter Goals:

- The basics of complexity and the need for self organization.
- Can’t organize around a system which is not understood.
- Nearly every organization suffers silo-like dysfunctions, and big picture learning won’t hurt.

3. Pretending to solve the problem writing software - 50%

After examining a recurring set of organizational and business dysfunctions, it's time to have a look to the way these problems are approached in the attempt of writing software .

Software engineers often think they are on a mission. Software is the backbone of company operations, it's the glue that enables successful businesses. Software solved a lot of problems to make things go faster, and smoother, compared to the cumbersome manual procedures that used to be in place. And disruptive businesses platforms could not be imagined without software[1].

Well, that's not entirely true. Our software has solved some problems, but chances are it also just seeded another problem somewhere else. The goal of this chapter is to understand *where*, *when* and possibly *why* software developers and architect often end up making a mess, even with the best intentions.

It's not about 'delivering software'

The original sin is a fundamental misconception about the true nature of software development. Software developers have been trained and paid according to the notion that the only valuable outcome of our work is the deliverable: the working software.

Developers often tend to see themselves as builders, craftsmen, artisans. This is tempting and powerful at the same time. Developers produce things that never existed before and completing a working piece of software that didn't exist, is an awesome - and *addictive* adrenaline boost.

---

[1] Can you imagine the paper-based version of a service like Uber? Or Ebay?

However, despite all the complex skills and the mastery of tools needed in order to write good software, *coding* is only the top of the iceberg. As the name “software solutions” may hint, *understanding the problem* is in fact the key game changer in enterprise software development[2].

It’s actually closer to music: *playing the song* is not exactly that hard, for an experienced musician, but *writing it*, well… that requires talent!

To put it in a tweetable format:

Software development is a learning process,

working code is a side effect

The person who said it best is actually Dan North in his Blog article *“Introducing Deliberate Discovery“* clearly pointing out that learning is the real bottleneck, in enterprise software development.

It’s not the *typing*, it’s the *understanding* that matters. However, once you acknowledge this, a whole world of inconsistencies starts unfolding.

We’ve been optimizing the wrong thing!

We can now deploy serverless components on the cloud in minutes. This is astonishing, if we remember that software development started from punch-cards. We are orders of magnitudes faster in *writing code*, and of course we used this speed to insert more layers of code in the stack, moving from room-size physical computers to distributed architectures.

But at a given moment, we kept improving the coding speed, even when it was no longer the key constraint. Having a last generation IDE with awesome code completion tricks is cool[3], but it won’t get much closer to delivering good software if we can’t access the right informations.

---

[2] With most of the solutions falling into the *‘not so exciting, once you understand them’* space. More on this later.

[3] May VI and EMACS lovers forgive me, I like comfortable IDEs. #FlameWar

On the contrary, it is embarrassing to notice how little we improved in terms of optimizing the learning side. It is still the norm in large enterprise software to have *segregated software developers* with only limited access to first-hand feedback and real users, and a plethora of intermediate roles to act as proxies. Learning isn't really supposed to happen.

Couldn't you just implement the specifications?

To be brutally honest, most of the common practices for organizing enterprise software development are *crap*. They focus on optimizing production, instead of learning. But this is just as dumb as somebody trying to lose weight just by *losing weight* without any look to daily habits, food consumption, sport activities and so on[^CAAIAO].

> *What is the value of code written on time, and on budget by someone who doesn't understand the problem?*

This is both a problem - tons of resources are wasted in huge software project that will never provide the value they're expected to deliver - and an opportunity. The moment we start working around the *real* problem, we might have a chance to significantly improve the situation.

The problem with the 'invisible deliverable'

There is a good reason why the builder metaphor is so tempting and widespread. Craftsmen and builders have an incredibly rewarding feedback loop: they *build something tangible that solves problems*. Think about a bridge. A well placed one will enable exchanges between nations or simply save time to commuters, for decades or centuries. Even if you were contributing just a little, you'll be *proud of your contribution*[4]. Some software can be just as rewarding, even if the more lasting bits of it, are buried in the deepest layers, far away from the attention of the man on the street.

---

[4] In practice, *pride* is a key ingredient for successful software development. Though it can turn evil, just like *the force* does in Star Wars, removing pride from our daily job is a recipe for failure. Daniel H. Pink illustrates this clearly in his book Drive.

Being *learners* instead of builders isn't that easy. Talented developers are often compulsive learners. Curiosity is their main driver, and they take pride in *solving the puzzle*. However, putting learning at the centers of everything isn't as actionable as we'd like.

- Unlike with software, there is no clear criteria or *definition of done*. When are we done learning? When is our learning enough? When should we stop learning[5]?
- How can we *measure* learning? How can we track our progress?
- How to treat the learning? Is it an *asset* or a *liability*?
- What happens to the learning after the software is released? Is the learning captured in the software, documentation or …*anywhere*?
- What happens to the learning that didn't make it into software?
- What happens when people that had a deep understanding of our problem domain leave our organization? Is that learning gone forever?

Just measuring deliverables is a lot easier. But this oversimplification is poisonous. The value for the company is not in the software itself, it's in the company ability to leverage the software in order to deliver value.

[FIXME: there is still something missing, internal experts and speeds of execution. Plus Learning organization.]

The volatility of learning

From a management perspective, learning is quite annoying. Mostly because, if you do the mind exercise of putting learning at the center of the things, chances are you'll have to admit your organization screwed it up big time.

Did you hire contractors for a strategic project? What happens next? If code is the deliverable, you're fine[6]. But if learning is the deliverable, you just set up collaboration in order to be sure that *learning will vanish at the end of the project*.

---

[5] The answer to this one is easy: *"Never!"*

[6] No, you're probably not.

The cost saving is actually only apparent. What happens like a reasonable strategy, is actually the shortest path to dissipation of a company competence.

The illusion of the underlying model

Nearly every person involved in writing software has a hidden agenda: they want to do a good job. This doesn't sound that scary, in fact is pretty well aligned with the famous Daniel Pink triad of *autonomy*, *mastery and purpose*; so this shouldn't be a problem.

The problem with programmers striving to do a good job, is that for the vast majority of them, *they have no experience of what a good job looks like*.

[FIXME: if all you do is fixing bugs, you're probably a bug fixer]

But spending too much time digging in the mud, and repeating yourself that you could have designed a better system *if you only had time*, is in fact distracting you from the danger of having this blind spot. Without a proper real-world example from the industry world, the only fall-back option available is *"what we've been taught in school"*.

Oooops!

Unfortunately, most of the problems my generation was facing at the university weren't even close to enterprise complexity. They were toy projects and they were lacking some of the key inf

What universities usually forget to teach

There are some common traits between how computer science is taught in many places around the world, that don't match the real world sills needed for software professionals. This has mostly to do with the educational format, and some institutions worked on the problem and improved a lot, but for the majority of them…

- Programming exercises are usually a matter of *turning requirements into code*: the assignment is written by the teacher and it's usually the same for every student. *But in the real world we might be required to do some real exploration of user needs, and requirements to code is so frustrating.*
- The assignment is supposed to be clear and easy to understand, in fact *ambiguity* can be a problem, and th. *But in the real world, requirements are **never** clear*
- The assignment is coming from a single teacher. *But in the real world multiple stakeholders with **conflicting needs** are the norm.*
- Exercises often start from scratch. No legacy previous attempt to solve the problem. No bad undocumented code to rely on or to blame in case of failure. *But in the real world, starting from scratch is a privilege reserved to a few, legacy code is usually the norm.*

Naive developers and analysts might have the illusion that the model is there, only the pieces of the puzzle are scattered around. You just have to find the pieces and put them together.

It's actually *fun* at the beginning. For a detail obsessed maniac like me, looking for clues in order to design a data model able to *represent reality* and to solve *more problems that it was originally designed for* was a rewarding secret pleasure. I was proving myself good, by *anticipating customer needs* and quickly reacting to change.

Unfortunately, it's not a *flat* puzzle where new bits of information fit together providing a consistent whole. In order to have a possibility to find a solution to a crucial problem we need to be sure that we're not pretending that the problem is something different from what it really is.

Here's the bad news.

The whole is not consistent.

As we discovered in the previous chapter, silos and specialization create a mess of independent models, each one serving a specific purpose.

Albeit not perfectly, this works in real life. People talk, and sometimes they understand each other. They might infer implicit information from the context (in a fashion show, a *model* is probably a good-looking person, in a developer meetup we'll be probably referring to some diagram instead) or have a long conversation with puzzled faces only to laugh when they discover they meant something different. They were suspecting it at a given moment, but this little unresolved ambiguity made the conversation more interesting.

Software doesn't work like that. Misunderstandings aren't funny, they're more likely mistakes that could cost a lot of money, or even human lives.

Coding and ambiguity don't play along very well. Coding is actually the moment when ambiguities are discovered, in the form of a compile error, an unexpected behavior, or a bug. Conversations tolerate ambiguities[7], but Coding won't forgive them.

Collecting nouns will lead you nowhere

It's then ironic to realize that software engineers have been tried modeling enterprise application looking for relevant nouns. Nouns are in fact the portion of the enterprise knowledge which is more prone to ambiguity.

Remember silos? Try asking: *"Do you know what an Order is?"* to different department like *Sales*, *Packaging*, *Shipping*, *Billing*, or *Customer Care*.

They're likely to agree if you show them the *static structure* of an Order (or the printed version of it). *An* `Order` *will have an* `Order_ID`, *a* `Net_Total_Amount`, *and a* `Gross_Total_Amount`. *The* `Net_Total_Amount` *will be the sum of the* `Unit_Price` *of the* `Line_Items` *times their* `Quantity`, *minus the applied* `Discount`. *It will be associated with a specific* `Customer`, *of whom we'll need to know both the* `Billing_Address` *and the* `Delivery_Address`. I stop here, we all know there's more.

You might find agreement on this definition. It will sound reasonable to everyone. Unfortunately, if you dig deeper in the system behavior you'll discover that:

---

[7] And *actively hide them too!* For many reasons, people would actually pretend they've understood even if they haven't. From the boss asking *"Is everything clear?"* in a threatening tone - obviously NOT, but who am I to attract the wrath and anger of a boss in bad mood? To the polite attempt not to disrupt the conversation, with a continuous request for clarification.

- In ***Sales***, a salesperson will *open a draft* order, *add* and *remove* items, and *apply discounts*. Then *place the order* once she has the customer agreement.
- In ***Packaging***, a warehouse clerk will *pick items* from the shelves, eventually *updating their availability* and *reporting missing items* in case of surprises. Once finished, the clerk will *seal* the package, and *declare it ready for sgipment*.
- In ***Shipping***, one or more deliveries will be planned for the given order. The order is *fulfilled* once all the items are delivered.
- In ***Billing***, we'll *send and invoice* for the order according to our billing policy, such as 50% open placing the order and 50% after order delivered.
- In ***Claims***, customers are opening a claim about a received order, if something didn't match the expectations: broken or missing items, and so on. Of course a claim can be opened *only* if you're the customer that placed the order, and only about the items included in the order.

[FIXME: finish this]

Expert modellers in this field are probably having the creeps due to my naivety. There's plenty of things I could have modelled more precisely, like explicitly separating `Orders` from `Packages` and `Shipments`

1. When learning new domains, these 'mistakes' are just right behind the corner. Given a new problem, tomorrow we'll probably get fooled again.
2. Multiplicity is one main driver to discover shortcomings of naive models. We'll need to separate Orders from Shipments because an order can take more shipments to be fulfilled, and so on.

We have a recurring problem in enterprise software development, which is knowledge unreliability, and a strategy for modeling (nouns and data first) that seems perfect for being fooled.

The Product Owner fallacy

If you embrace the idea that software development is mostly a *learning endeavor*, then some common practices begin to smell.

In Scrum, a lot of responsibility is placed upon the shoulders of the Product Owner, who serves as the primary interface between the team and the business stakeholder.

However, the interpretation of this key role is frequently at odds with the learning perspective. Often, teams are kept in isolation because the PO is taking care of all the necessary conversations with different stakeholders.

Slowly, the Product Owner becomes the only person who's entitled of learning, while the development teams turns into a mere translation of requirements into code. Everything starts to smell like the old *Analysts vs Developers* segregation that was so popular in Waterfall-like environment.

[FIXME: A picture of Product Owner like a man in the middle would help]

But if we think software development as learning, it suddenly becomes obvious that *second-hand learning is not the best way to learn*.

If your goal is to learn to ride a bike you can choose between:

- get a bike and try it,
- talk with a biker first,
- talk with a friend that knows a biker,
- read a specification document written by a friend that talked with a biker.

Choice is yours.

Single point of failures are bad

Placing the burden of understanding the whole system on a single person/-role sounds like a terrible idea even in strategic terms.

Checklist: How many people understand your system?

- Nobody –> you're probably screwed
- Only one person –> you're probably *even more screwed*, but it's harder to admit.
- A few people are reasonably competent on the whole –> safer.
- Everybody knows the whole story –> just lovely.

Re-framing the problem

What Scrum really pointed out was that a development team is very bad at managing conflicting ***priorities***, and it's a Product Owner responsibility to sort them out before the development iteration takes place.

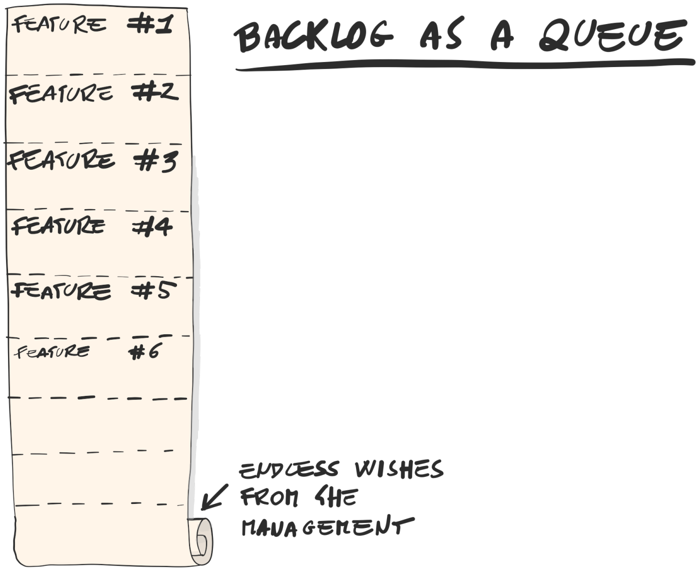
What Scrum **didn't prescribe** is that *all the learning should happen through the product owner*. This is a dysfunction that lowers the quality of the overall learning and of the resulting product.

]

The backlog fallacy [FIXME: definitely not the first one]

In 2016 I can probably assume that if you're involved in software development, you'll be doing some form of agile process, such as Scrum or Kanban, and that you're familiar with the idea of a Product Backlog, a collection of features to be implemented by the development team, iteration after iteration.

Once in place, a product backlog tends to look like a *queue*



Backlog as a queue

What is the problem with the backlog? Iterations are supposed to be a good thing.

What I don't like, is the fact that backlogs provide the illusion of *the whole project is just the sum of its parts*, which, unfortunately, isn't the case.

A backlog is optimized for *delivery*. Having a cadence and releasing in small increments works great in order to provide a heartbeat and a measurable delivery progress.

But a cadence in which every week repeats the same old basic schema of the previous one, with *planning*, *implementation*, and then *release* may not leave enough space for those activities that don't follow the plan.

In fact, some project follow the plan relatively well. They're usually the project where there's not that much of *discovery* to happen. Compliance projects are a typical example: a new regulation mandates certain requirements and we just have to deal with a checklist.

Boring

[FIXME: system sensitivity around the Bottleneck]

Breaking iteration habits

[FIXME: duplicate with next chapter. Choose a version!]

I've seen the insurgence of these habits becoming a burden towards development of meaningful products. Unconventional activities like 'going on the field to see how users are really using our software', or 'running a large scale workshop in order to better understand the problem that our software is supposed to solve', simply don't happen, because there's something more important, already planned for this week[8].

Habits are great in providing a healthy routine. But they create a lot of resistance when you need to change them, or to do something different. If your routine is *office - coffee - stand-up - open the IDE*, doing something different requires a lot of effort.

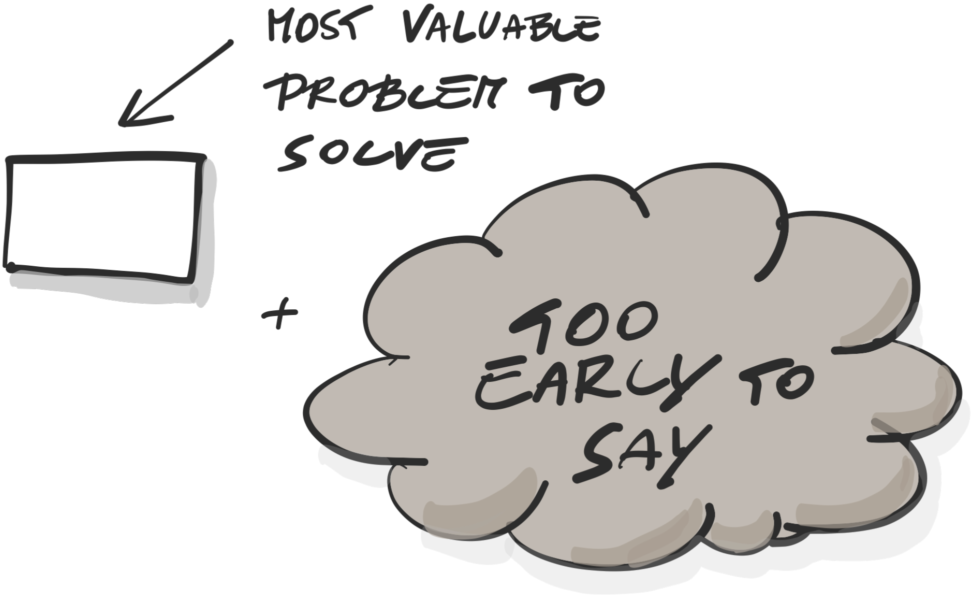
If repeatable weeks are optimized for *planning* and *delivery*. They aren't really optimized for *discovery*. In fact discovery is more likely to happen when we break our habits and do something distinctively different[9].

---

[8] Even worse, in retrospective you might find yourself saying: *"If only we found the time to go and see what users were really doing!"* While contemplating the money wasted on a feature that no one really wanted.

[9] The greatest introduction to this way of thinking is *Pragmatic Thinking and Learning* from Andy Hunt.

Embrace Change

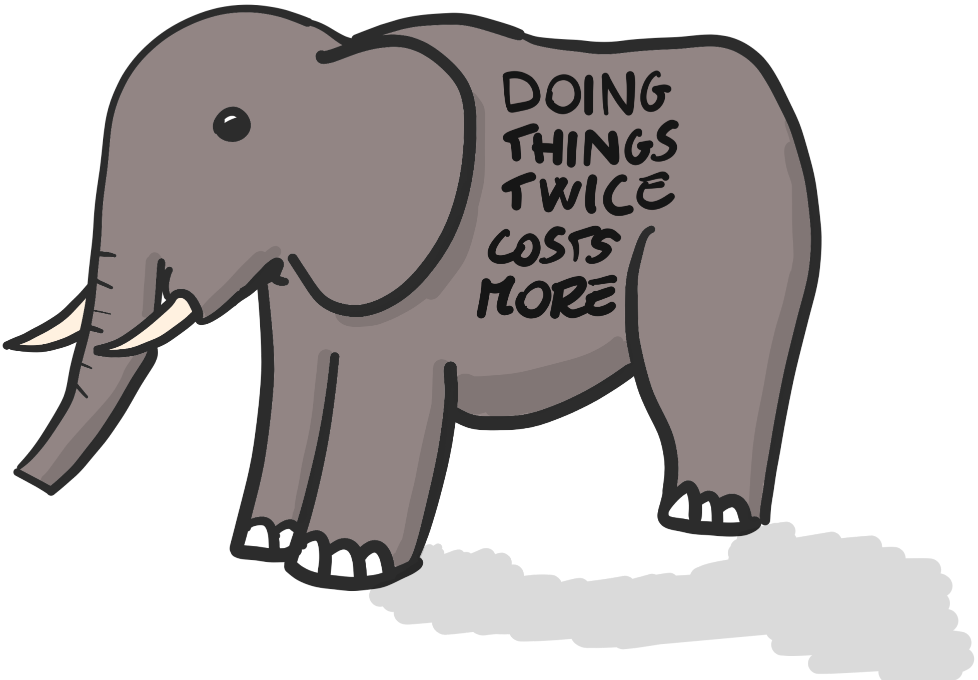

MOST VALUABLE PROBLEM TO SOLVE
+
TOO EARLY TO SAY

That's actually the backlog I like most

Embracing change is suboptimal

There's something in the whole thing of *embracing change* and of *iterative software development* that always looked like an elephant in the room to me.

Doing a thing twice costs more than doing it right at first shot.



No reason not to try to do things right FIXME: the elephant has only three legs.

Yes, I know. I am the same guy that talked about the backlog fallacy just a few lines above. But *iterative development* doesn't mean we shouldn't *try* to start right.

> I have a degree in electronics[a], which means I spent some time sweating on equations that had to be solved by *interpolation*: you choose a value for variable X, which leads to a value for Y. You then apply Y to get a more precise X, and repeat till the delta between the old variable and the new one is below a given threshold, showing that the system is converging on a given set of results. Well …I learned the hard way that *choosing the right starting point matters*: a *plausible* pair of variable was leading to the expected result in 2-3 steps, while a wrong starting point was taking way

> more rounds, hence more *time*, severely hurting my possibilities to pass the exam.
>
> ---
>
> [a] Totally useless, by the way. I hated the topic so much that I can't even have fun with toys like Arduino, now that they're cool.

End of the digression, but I hope the message is clear: *iterative development is expensive*.

It *is* the best approach for developing software in very complex, and lean-demanding domains. However, the initial starting point matters, a lot. A big refactoring will cost a lot more than iterative fine tuning (think splitting a database, vs renaming a variable). So I'll do everything possible to start iterating from the most reasonable starting point.

I can't promise this will work. I can't guarantee there won't be surprises along the way.

In fact, I'll be extremely happy to discover *breaktroughs*[10] to prove that my original assumptions are wrong, even if it means throwing away what looked like a brilliant idea.

I do my best to start in the right way just because *iterations are expensive* and the fewer, the better.

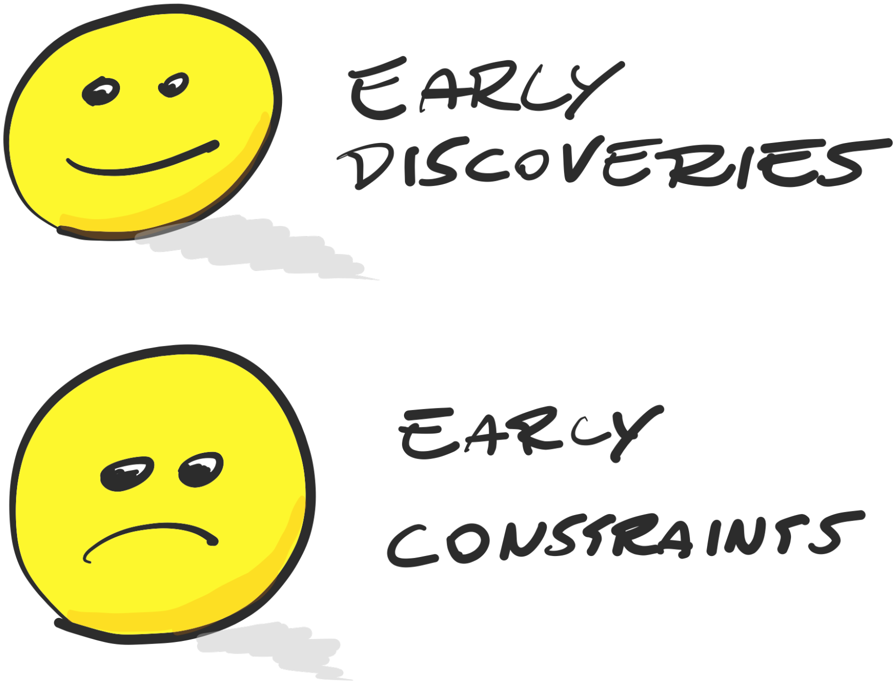
The revenge of the U* word?

Upfront is a terrible word in the agile jargon. It recalls memories the old times *analysis phase* in the worst corporate waterfall. Given this infamous legacy, the word has been banned from agile environments like blasphemy. But unfortunately …there's always something upfront. Even the worst developer thinks before typing the firs line of code.

This is not what EventStorming is for

---

[10] Dynamic Systems Development Method was one of the first approaches to Agile software development: https://en.wikipedia.org/wiki/Dynamic_systems_development_method



There is no reason not to anticipate learning, it's adding constraints assuming that we already learned everything that creates a mess

What about emergent design?

[FIXME: candidate for removal. Or for a box.] Somebody once told me *"so you're against emergent design?"* ...well I am not. I just think that emergent design is a great tool to help you find your way in scenarios of great uncertainty. If you have no idea about how the outcome is going to look like, then emergent design is the tool for you.

But this is not the case when it comes to model business processes. Despite companies describing themselves like *unconventional*, processes won't be that different. Or more precisely, *process building blocks will be exactly the same*. Rediscovering them from scratch is just like watching a monster movie, pretending that we don't know already who's gonna be the hero that survives.

It might be fun to apply emergent design principles to a problem that has already a solution, but it's flushing money in the toilet.

Enter Domain-Driven Design

Among all approaches to software development, Domain-Driven Design is the only one that focused on *language* as the key tool for a deep understanding of a given domain's complexity.

Domain-Driven Design doesn't assume consistency of the different areas of expertises. In fact, it states that consistency can only be achieved at a model level, and that this model can't be large.

When modeling large scale system, we shouldn't aim for a large "enterprise model". That's an attractor for ambiguities and contradictions. Instead we should aim for 2 things:

- multiple, relatively small models with a high degree of semantic consistency,
- a way to make those interdependent models work together.

A small, strongly semantically consistent model is called Bounded Context in Domain-Driven Design. That's basically

A good old-fashioned Bounded Context example

I am Italian. According to the stereotype, I should love coffee. The problem is what I call *'coffee'* is probably not what you'd call *'coffee'*, unless you're Italian too.

When talking about coffee, everybody pretend they understand what you're talking about.

But if you drill down into the implementation, you'll discover that *'coffee'* for an Italian usually maps into what foreigners call *'Espresso'*. Something to drink *quickly* in a small *cup*, while *standing* at the bar. In other countries (particularly colder countries), a coffee is associated with a *mug* and the idea of drinking it *slowly* while *sitting* at the table, or even a working desk.

> This is what I ask for when I am traveling: *“An espresso, please.”* This is not what I ask when I am in Italy. In Italy, I just ask for *“A coffee, please.*
>
> If I ask for an espresso in Italy, I get exactly the same thing, plus a raised eyebrow from the barista: *only foreigners call it espresso*.
>
> If I am looking for precision - and working software doesn’t really like *ambiguity* - it’s my job do define the boundaries of conversation that can happen without any doubt about the real meaning of the terms used.
>
> A **Bounded Context** is exactly that: a portion of the model which we must keep ambiguity free. Every word in the model has exactly *that precise meaning*.

[FIXME: context map example]

The context map will resemble the areas of expertise. But there’s a crucial difference: we can’t assume internal consistency in an area of expertise, while that’s our primary goal within a bounded context.

The backlog fallacy (rewritten)

Agile approaches such as Scrum and XP tell a lot about how a software development project should be managed by turning the stakeholders expectations into *Features* or *User Stories* and splitting delivery into a sequence of *iterations*. I actually love this stuff. But I still have a question mark.

My question mark is about *“how do we start?”* I actually like the fact that there is no strong *prescription* about how to initiate an agile project,

Delivery cycles …really?

What I do like about iterations

- **Frequent feedback**: for a team needing to understand whether they’re on track or not, *feedback is the highest value currency*. This is what they need, whether they like it or not.

What I don’t like about iterations

- **Iterations tend to repeat**: team estimate, based on velocity. Velocity is calculated on past iterations. This is an implicit driver to make an iteration comparable to the previous one, maybe just to show that we improved a little.
- **Little space for doing things differently**: there is a strong driver to turn frequent delivery into a *habit*. This is powerful, but suboptimal. There are iterations where you should really do something different than Sprint Planning, design, implementation, delivery, demo, etc. Things like *“Hey, this week we took a couple of days on the field to see what users are really doing with our software.”* …aren’t really in the cards.

Let’s be more explicit: repeating things week after week is *boring* and

Boredom is the arch enemy of learning[11]

If learning is our primary goal, the we should systematically remove every impediment on the way of better learning, and repeating empty ceremonies, iteration after iteration isn’t probably the best idea.

Modeling is broken

A model is just a tool towards deeper learning

---

[11] The sentence above is valid in schools too. *Being boring* is the ultimate sin for a school teacher, especially when one can count on an infinite source of curiosity, like a kid’s brain.

Requirements gathering is broken

Enterprise Architecture is broken

If we're thinking large scale, then sooner or later the words Enterprise Architecture will start peeking out of discussion.

Treating it like an ecosystem

Data-based modeling is resilient

Don't take the statement above as a compliment. It's not. It's just a matter of entropy.

> *Take two boxes of Lego. Open them. Now mix the pieces. Now separate them.*

[FIXME: there's something in the middle missing]

Does it feel stupid? Weird, because I've just described the most common enterprise architecture on planet Earth, which is basically (in technical terms) *a database filled up with ambiguously defined data, plus some layers on top of it*.

There are historical reasons why this architecture is so common. And right now engineers are getting a degree from universities without having a clue about why this approach is wrong. However, in statistical terms this is the most common architecture.

Model Splitting is expensive

Splitting established model in a Database-centric architecture is among the most expensive refactorings in software development. Quite often the risks are so hard to evaluate that initiatives are killed before the start.

Even teams embracing agile, these refactorings tend to float in the backlog in a loop of continuous procrastination, while bugs emerging from *nobody knows where* always get the top priority lane.

I have no definitive solution for this problem[12].

The EventStorming approach

There's a connection between the different problems we've just highlighted. If you remove the context from the picture it all boils down to a very few distinguished things.

1. *See the system as a whole.*
2. *Find a problem worth solving.*
3. *Gather the best immediately available information.*
4. *Start implementing a solution from the best possible starting point.*

This is what we do in EventStorming: we gather the best available brains for the job and we collaboratively build a model of a very complex problem space.

* * *

---

[12] But I have some ideas indeed. If you're interested, you might want to take a look to an old presentation of mine: http://www.slideshare.net/ziobrando/drive-your-dba-crazy-in-3-easy-steps

4. Running a Big Picture Workshop - 98%

In the last two chapters, we saw how many organizations end up sharing the same dysfunctions, and how there is no easy way out without a shift in the mindset. In fact, most of today's problems originated from yesterday's solutions.

Writing software on top of the existing dysfunctions can only make things worse. The more expensive the software, the harder it will be to dismiss it later, and the undesired consequence is that the next needed change would be harder and more expensive.

In a perfect, imaginary, organization, everybody would have a clear idea of what they're doing and they would be able to provide the right information to the software development team in order to have the best possible software delivered.

In practice, no organization is flawless. Despite official claims, processes, people, technology, and tools will have weaknesses and holes and will need improvements.

Pretending to solve the problem, assuming that the rest of the system won't change in the meanwhile, is simplistic at best, and dangerous in most cases.

In a changing world, everything is evolving: technology, people, the organization itself, the surrounding business and the job market too. Assuming that one of these ingredients will be immutable during a nontrivial change initiative is not a legitimate simplification: it's just plain *naive*.

But there is one thing that we can do: we can take a snapshot of the current reality and ensure that all the key players are looking at the same thing.

Whether the goal is to write a piece of critical software for the organization, or to design the next implementation of a business flow for a startup, or to redesign a key business process for an existing line of business, this means having *a clear, shared understanding of the problem landscape and of the possible solutions*.

Our recipe to achieve this result is really simple:

Gather all the key people in the same room and build together a model of the current understanding of the system

Unfortunately, decades of boring dysfunctional meetings have taught us that just gathering people, without a specific format, will lead you nowhere. Good news are we *do* have a format for addressing this problem, and it's called **Big Picture EventStorming**.

A Big Picture EventStorming is one single large scale workshop that involves all the key people that we expect cooperate in order to solve critical business problems.

If *understanding the system* is the key ingredient to choose promising solutions and eventually implement the right software, then learning as much as we can, before wasting money on the wrong initiative, is the way to go.

In this workshop:

- we'll build a behavioral model of an entire line of business, highlighting collaborations, boundaries, responsibilities and the different perspectives of the involved actors and stakeholders;
- we'll discover and validate *the most compelling problem* to solve;
- we'll highlight the major risks involved with the status quo, and possible new solutions.

Key ingredients for a Big Picture EventStorming

Here is a list of the key ingredients for a successful workshop.

- *invite **the right people**, we are looking for the perfect blend of curiosity and expertise, bound by the common goal of improving the system;*

- *a suitable location, possibly a room large enough to provide the illusion of an* ***unlimited modeling space****;*
- *at least* ***one facilitator*** *in charge of providing guidance and making sure everything runs smoothly;*

That's it! And since participant's time is precious …everything will have to happen in just a few hours!

Now, let's try to describe the steps needed to run a great big picture EventStorming session. Of course, there will be plenty of tips about how to put it in practice, and explanations about why this approach works, in order to make the magic happen.

Invite the right people

In order to run a successful workshop, you'll need to have the right people on board: a good mix of knowledge and curiosity, but most of all you'll need *people that actually care about the problem*.

Diversity in background and attitude is crucial. In fact, EventStorming provides a solid background for meaningful conversations across silo boundaries, and an adaptable format (see *fuzzy by design* and *incremental notation*) that allows a collaboration across multiple disciplines. This means that business experts, lean experts, service designers, software developers can all be part of the same conversation.

Getting all these people together in the same room at the same time can be itself a hard mission to accomplish. We'll talk about it in details in the managing invitations section, in the next chapters.

Room setup

We'll need a special room arrangement for the workshop.

A typical corporate workshop involves about twenty people, but numbers can rapidly grow. Participants won't be sitting around a table, they'll be moving around in order to have conversations with different people facing different portions of the modeling space.

Startups will involve fewer participants, have different workshop dynamics but will use roughly the same amount of modeling surface.

In order to let the magic happen, we'll need to hack the space on our favor and provide an environment with no impediments to the ideal workshop flow. We'll talk about this topic in detail in the prepare a suitable room section.

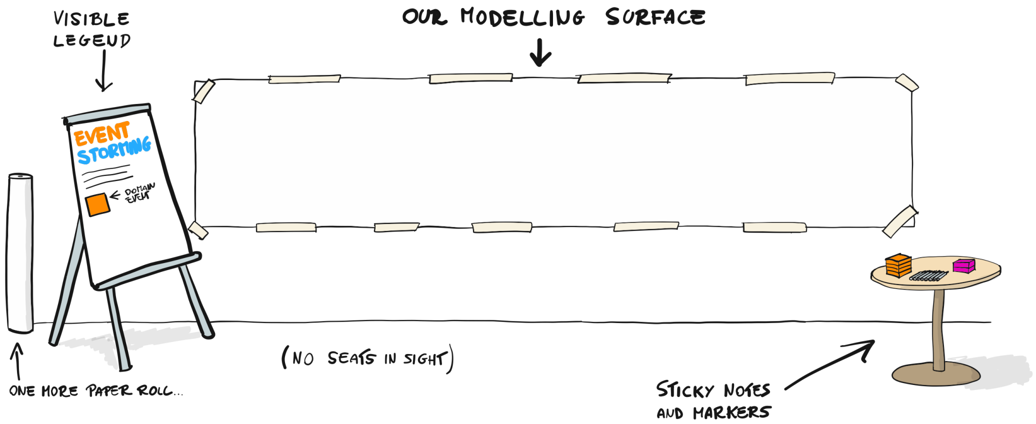
For now, let's just assume that a few criteria are met.

- The room has a **long straight wall** where our paper roll can be placed as our modeling surface. 8 meters is the minimum. The more, the better.
- There's **enough space for people to move** by the modeling surface. People will need to see the forest and the trees.
- **Seats are not easily available**[1]. They'll be needed after a couple of hours, but they're just terrible at the beginning. Stacking them in a corner is usually a good enough signal.
- The **paper roll** is put in place on the long straight wall.
- There is a flip-chart or an equivalent tool to be used as a **legend**, for the workshop.
- There is **plenty of sticky notes and markers** for everyone.
- There's enough **healthy food and beverages** to make sure that nobody will be starving.
- There is a **timer**: some phases will need to be time-boxed. A visible clock might come in handy.

Before the workshop, your room should look more or less like the picture below.

---

[1] Be ready to handle exceptions: some people might have very good reasons to need a seat. My policy is to *be soft on needs, be hard on laziness.*



Good summary of a room setup

Everything is in place? Great job! That's all you need to get started.

Workshop Structure

The action will take place in phases of increasing complexity. We'll keep things easy at the beginning, adding more details as long as people are getting confidence with the format.

We'll leverage the idea of incremental notation in order to keep the workshop in a perennial *"Goldilocks state"*: it has to be *not too challenging, not too easy, just right!*

In order to give you the exact feeling, I won't anticipate the structure or the list of the phases. But I promise you'll get a good summary at the end of this chapter.

Phase: Kick-off

Room is ready. Participants are in. It's time to kick-off the workshop.

I usually start the workshop with a quick informal presentation round, in order to discover everyone's background, attitude and goals, and to allow everyone to introduce themselves. It has to be quick (because a boring round robin of all participants can be really expensive) but I suggest not to skip this part: a wrong check-in can actually backfire later[2].

You might want to **explicitly set the goal**: *"We are going to explore the business process as a whole*[3] *by placing all the relevant events along a timeline. We'll highlight ideas, risks and opportunities along the way."*

I do warn participants in the beginning about things that can make feel them uncomfortable: the workshop is going to be *chaotic*, mostly stand-up, it's going to feel *awkward* at given moments, *and this is all expected*.

It's going to be unusual for most of the participants, we'll be taking them out of their comfort zone, and they have to be reassured that they won't be doing anything wrong. Keep in mind that the more workshop you run the more confident you'll get, but there will always be a person in the room participating for the first time.

I also inform everyone that my explanation upfront is going to be very short. Some people tend to expect very long introductions and detailed instructions. The distinct flavor of boredom at the beginning of a dysfunctional meeting can be reassuring for some. Others, maybe used to *lead* those meetings, might find incredibly annoying not to be in the driver's seat[4].

I personally find more productive to get in the action as soon as possible. This usually means that somebody will feel like 'thrown in the swimming pool', your job is to communicate the *“Don’t worry, I am here to help!” feeling. They’ll be happy in a while, but, well… _nobody likes to be thrown in the swimming pool!*

---

[2] If participants don't know each other at all, then an explicit ice breaking game might be necessary. If they already know each other… we still need to allow every participants to introduce themselves, and to make their personal status and expectations explicit, you may also want to check the usual suspects as an icebreaker.

[3] I am reluctant to fall back to the usual *from the beginning to the end* because, as you'll discover during exploration, the whole idea of beginning and end in a business process is a myth. Feedback loops and returning customers are what keeps your business alive.

[4] We'll talk about the godfather behavior and how to deal with it in the anti-patterns section.

My friend Paul Rayner likes to run a time-boxed warm-up exercise, by modeling together a well known story (*Cinderella* is one of the favorites) so that participants get familiar with the basics of the method, without worrying of their problem space first.

In short: take care of the people’s feelings. It’s not machinery, it’s *people*. You won’t regret it.

What I don’t do now

Despite how much I like my creature, I try hard not to pitch the method. EventStorming is cool, but even in the trendiest startups, it’s just a tool, not the final goal. I don’t even explain the method much before the action: every explanation calls for one more question that is stealing time from the workshop goal.

So …don’t talk, *show*.

Make it happen.

If you do a good job, you’ll get plenty of questions at the coffee machine, later.

Phase: Chaotic Exploration

The first step will use the simplest possible notation: we'll name orange sticky notes *Domain Events* and place them along a timeline in order to represent our whole business flow.

Most of the people in the room won't be familiar with the concept of Domain Event, so a little explanation would be needed.

Usually it all boils down to three basic concepts:

1. it has to be an **orange** sticky note[5];
2. it needs to be **phrased at past tense**, like *item added to cart* or *ticket purchased*;
3. it has to be **relevant** for the domain experts.

Everything you need to know about domain events

The term *"Domain Event"* comes from software architecture, and isn't the the most inclusive one in some situations. I sometimes call it simply *"Event"* and it works fine.

The Event will be the building block of our business-related storytelling. We'll build our narrative as a sequence of related events.

---

[5] Why Orange? It's entirely arbitrary, but sticking to a consistent color convention will make every picture more readable. The whole story may be read here.

Using a verb in the past tense will sound arbitrary to some. There are a few good reasons to use this phrasing, but they won't be evident before the action takes place. I try to play variations of *"trust me, I know what I am doing"* not to get swamped in long explanations before the action.

Making the notation explicit from the beginning in a visible legend will surely help.

Getting into a state of flow

I expect the first minutes to be awkward: most people won't know exactly what to do, and will look around, in order to get some clue from their peers, in a stalemate situation.

This is also the hardest moment for the facilitators, people will look at them for guidance, but their job is to support, not to actively lead.

An icebreaker, the person that places the first sticky note somewhere in the middle of your modeling surface, is your best ally. Their first sticky note will signal to everyone else in the room: *"It's not that hard. It's really just writing a sticky note, with a verb at past tense and place it somewhere."* But even more important is the implicit message: *"You can do it too. Nobody is going to get hurt.*[6]*"*

When this happens, praise the icebreaker. Rewarding bravery will show other participants, that *just doing it* is the winning behavior in this workshop.

If there is no bold icebreaker in sight, it might be the facilitator's duty to get past the deadlock, placing an example Domain Event somewhere in the middle. I tend to resist it, since it may put other's participants in passive mode, and then rely too much on the facilitator for the next steps. If I do, then I move away immediately from the modeling surface: *"Now it's up to* ***you****, not me."*

Once the ice is broken, the workshop ignites[7]. It's funny to see how it turns into a chaotic, massively parallel activity where everybody is contributing to the model at a surprising fast pace.

---

[6] it is interesting to note how kicking off the workshop is a lot harder in corporate scenarios. Startups are usually a piece of cake, culture is usually very experimental and there are not so many past mistakes to worry about, but in corporate… you can tell there are a lot of scars and past blames in the room, which are implicitly suggesting: *"let someone else make the first move."*

[7] Dan North once compared kicking-off an EventStorming to starting an old chainsaw: …*wrrrr*, …*wrrrr*, and then all of a sudden …*WRAAAAAAAAAAAAAAAAMMMM!!!*

Rephrasing to use past tense

I don't expect the initial prescription about the verb at past tense to be followed by everybody. It's declared at the beginning, and visible on the legend, but I try not to stress participants about it. Some people have a hard time re-framing their internal narrative in terms of verbs in the past tense, and confidence and engagement are more important than precision or compliance to an arbitrary standard, at this stage. So, I might just let it go for now and come back on the notes later[8].

[FIXME: picture]

However, a different phrasing, using an active verbal form like `Place Order` instead of `Order Placed` is a minor issue at this moment, while using phase names like `Registration`, `Enrollment` or `User Acquisition` will filter out too many details from the model. Some people are tempted not to dig deeper into those phases, but *we* do care about the details.

We provided an unlimited modeling surface *exactly to be able to see these processes at the right level of detail*. And the right level is Domain Events. Phases will eventually emerge later anyway.

Growing the timeline organically

During the chaotic phase, there's no common denominator on people's behavior. In fact, observing the way people self-organize for the modeling activity can tell a lot.

- Some might form small committees trying to agree on a common phrasing. Facilitator should politely step in and break the ring, since discussing to reach an agreement on every single sticky note, before writing it would kill workshop throughput and hide *exactly the contradictions we want to explore*.

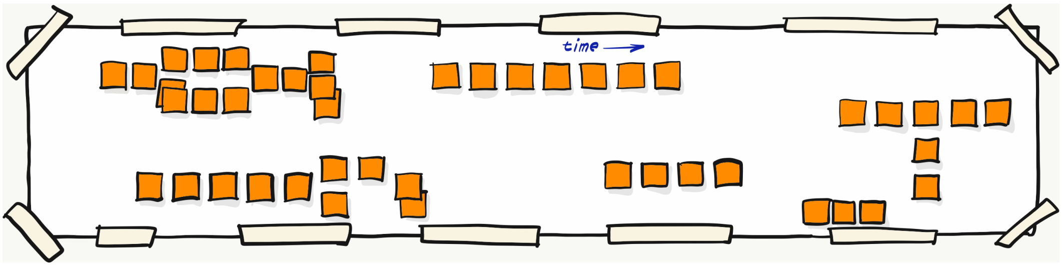
---

[8] Most of all, I try not to use the word *wrong*. Keep in ming that you're asking professionals to get out of their comfort zone, to try to model their own business in a counterintuitive way. Punishing them for a genuine effort can backfire.

- Some might work mostly alone, dropping the bulk of their personal expertise in a single strip of orange sticky notes, ignoring the surrounding world.
- Some might have no idea about what to write, and they'll need to be reassured that *guessing* is a totally legitimate action in EventStorming.

I am not expecting many conversations at this stage. After breaking the committee circles, people will eventually start working on their own: I call this phase *quiet chaos*. The model starts growing quickly and organically in a loosely controlled fashion.

The resulting model will usually be big and messy. Dozens of sticky notes, maybe hundreds. Some duplicated, many not in the correct order, plus some unreadable ones. No worries, we weren't aiming for perfection.



Different actors might have created different locally ordered clusters in a disordered whole

More precisely, I am expecting to have *locally ordered clusters in a disordered whole*, and the timeline constraint to be broken in a few places.

No problem, we'll sort out this mess in the next step.

Cool down

Eventually, the crowd will stop adding stickies to the wall and will take a more contemplative position, looking at the big picture more than to their own stickies, and walking a few steps back[9].

This is a good moment to praise participants for the great result, allow for compelling questions, and to take a break. The next steps will require a lot of energy.

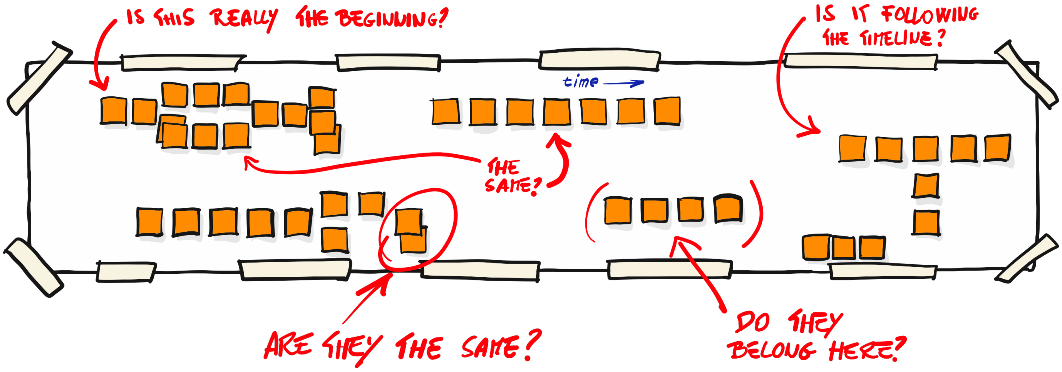
---

[9] The typical Chaotic Exploration phase is usually a sequence of *panic (the standoff)* -> *icebreaker* -> *quiet chaos* -> *contemplation*.

Phase: Enforcing the timeline

During *Chaotic exploration*, we asked participants to place domain events along a timeline. We shouldn't expect the outcome to be perfect.

If the exploration phase has been chaotic enough, you should now see a rough timeline, with duplicated sticky notes, and locally ordered clusters hidden in a messy flow.


IS THIS REALLY THE BEGINNING?
IS IT FOLLOWING THE TIMELINE?
time
THE SAME?
ARE THEY THE SAME?
DO THEY BELONG HERE?

Different actors might have created different locally ordered clusters in a disordered whole

The more parallelism (which is *good* for speed and information), the messier the result. Experts may just take a marker and write down everything they know about the domain, and place everything on the wall in a single batch, while the ones trying to keep the whole timeline consistent will give up at some point.

Room layout plays a key role too: not enough walking space in front of the modeling surface, means that people wont easily move around, and will focus on performing their task first, and look at the whole later[10].

Now our goal is to make sure we are actually following the timeline: we'd like the flow of events to be consistent from the beginning to the end.

This is when the discussion gets heated: local sequences - *"this is how it works in my own silo"* - have to be merged with somebody else's view on the same event. And the whole thing needs to make sense. Inconsistencies start to get visible, and once they're visible …somebody will talk about them!

---

[10] Room layout has a disproportionate weight on the outcome of the workshop, even the best facilitators can't beat lack of oxygen or walking space, so don't underestimate room picking.

Key conversations usually just happen. This is a moment where your team's ability to self-organize may surprise you, but facilitator's guidance might be needed in order to provide some structure.

Provide Structure

Just 'sorting out the events' in order to enforce the timeline isn't as simple as it sounds. Participants will try to swarm over the modeling surface, attacking *that sticky note that didn't fit*, in a rightful attempt to sort out the mess, only to discover that brute force isn't the most efficient strategy.

Make sure there's enough space available

The main impediment to efficient sorting is the availability of empty space: sorting out a surface with too little slack becomes a puzzle on its own, and we don't need that extra complexity.

This is a good moment to add more space, maybe sticking one more paper roll below the original one, if the orange-to-white ratio is too high. If that option is not available, you may want to look at *Chapters Sorting* in a few pages.

Choose a sorting strategy

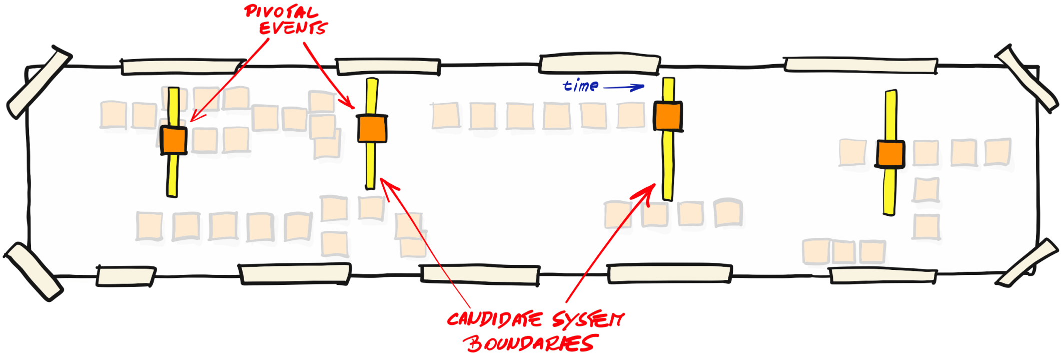
Brute force approaches to sorting don't work very well on a very large messy set, sorting works better with more sophisticated strategies. There are a few recurring ones: *Pivotal Events*, *Swimlanes*, *Temporal Milestones*, *Chapter Sorting* and *The Usual Suspects*.

You might want to start picking the most promising one, given your context.

Pivotal Events

With **Pivotal Events** we start looking for the few most significant events in the flow. For an e-commerce website they might look like `Article Added to Catalog`, `Order Placed`, `Order Shipped`, `Payment Received` and `Order Delivered`. These are often the events with the highest number of people interested (an `Order Placed` can trigger reactions in the billing department, and in shipping too, not to mention fraud detection or loyalty points).

My favorite tool here is a colored label tape, that I place below the pivotal event sticky note, as a *boundary* between the different phases of the business flow.



Highlight pivotal events to allow simpler sorting of what's in between

I wouldn't spend much time looking for a perfect choice of pivotal events. Bear in mind that at this moment we're just trying to speed up the sorting operations, so preserving flow is more important than reaching consensus.

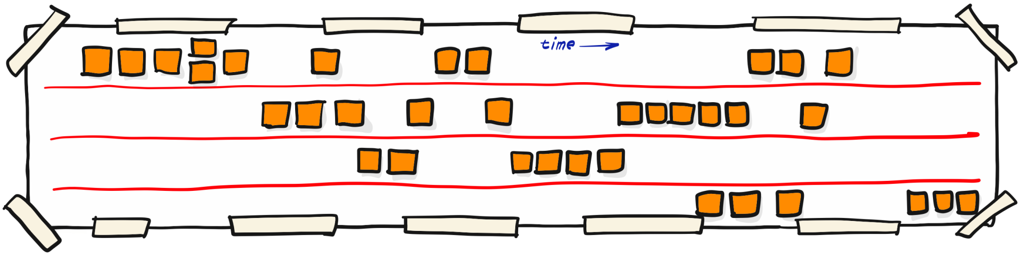
Usually 4-5 key events are enough to allow quicker sorting of the remaining events[11]: people will easily place their flow inside the corresponding portion of the flow, and eventually start conversation with their neighboring peers.

---

[11] if you have a background in software development, this might feel like shifting from *Bubblesort* to *Quicksort*.

Swimlanes

Separating the whole flow into horizontal **swimlanes**, assigned to given actors or departments is another tempting option (this seems the most obvious choice, if you have a background in process modeling) since it provides better readability.



Horizontal swimlanes provide better readability, but they end up using a lot more space.

Unfortunately, readability comes at a price: every swimlane will occupy a significant portion of the vertical space, which is limited, .

we'll never have only 3 or 4 swimlanes. We're not mapping a single process: we're mapping many processes at the same time. This means:

1. more actors involve, and so more swimlanes to be stacked up vertically;
2. more white space along the swimlanes because of the synchronization between the flows.

In general, swimlanes play very well for a single process, or for a few distinct processes that tend to happen in parallel to the main business flow.

For us, this means that it's often more efficient to apply swimlanes after a temporal structure has been established, with *pivotal events* or *temporal milestones*.

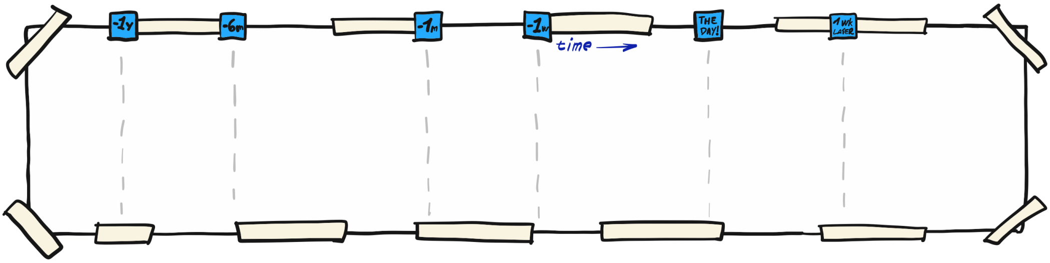
On a smaller scale, I like using horizontal labels to name the parallel processes. A little less readable than real swimlanes, but more space-efficient.

Temporal Milestones

For some businesses, establishing key events may not be the best option. There might be many concurrent processes (organizing a conference is a typical example) or too much misalignment about *what comes first*. In these situation you may favor **temporal milestones**: making the temporal frames *roughly* visible on the surface so that everybody can place their items *more or less* where they belong.

I usually place blue sticky notes on top of the modeling surface with some typical time indicator like `1 year before`, `6 month before` `1 month before` and so on (usually choosing smaller intervals the more we get closer to the core of the process).

This helps placing events *more or less* where they belong. Keep in mind that we don't know yet how much space should be used for any given activity, so be ready to shift the milestones or to replace them if you discover something more precise.



Time might be a better splitter than events in some cases

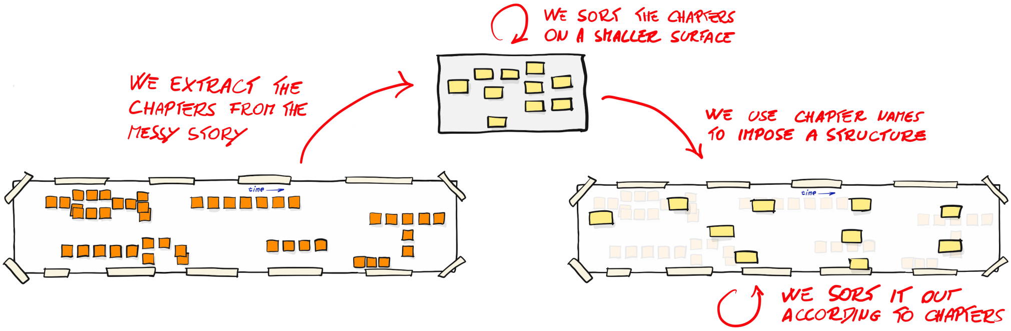
Of course, we're not interesting in the trivial ordering. What we do care about are the conversation that emerge from the not so trivial ones.

Chapters Sorting

How do we sort out things if there's too little maneuvering space? Like too many orange stickies and too little slack space and no possibility to add more space?

My last resource is **chapters sorting**: instead of sorting out the whole thing, we do this instead.

1. We run a quick *extraction round* of the key Chapters[12] of the whole business story. This will usually lead to 15-25 chapters (usually I write them on larger yellow stickies).
2. We quickly sort the chapters on a different surface (often a window), resulting in a skeleton of the desired structure of the flow.
3. We then apply the chapters structure on the main flow and remodel everything accordingly.


WE SORT THE CHAPTERS ON A SMALLER SURFACE
WE EXTRACT THE CHAPTERS FROM THE MESSY STORY
WE USE CHAPTER NAMES TO IMPOSE A STRUCTURE
time
time
WE SORT IT OUT ACCORDING TO CHAPTERS

Chapters sorting in action: sorting a few chapters on a clean surface takes less energy than sorting the whole thing. Once we see the structure we can apply it on the real model.

"But ...why didn't we start from chapters from the very beginning, before being swamped with events?" Well ...we couldn't be *sure* about the chapters, before the chaotic exploration.

Put in another way, I just don't trust the official version.

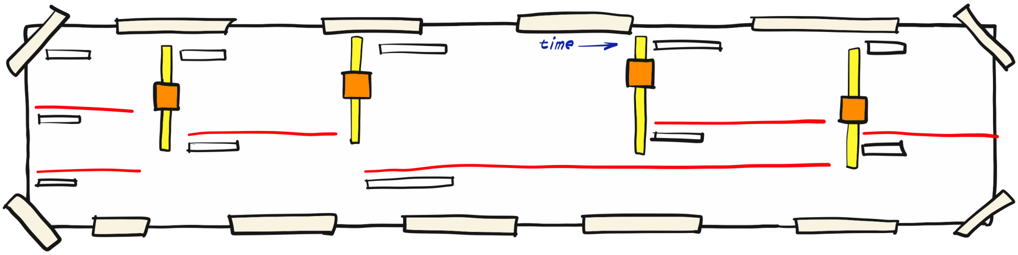
---

[12] Why do I call them chapters? Reason number one is that I am trying to use the consistency of the business storytelling as a gluing factor.

Combining multiple strategies

It should be clear now that a single strategy is not enough to rule the complexity of the whole thing, and that you'll have to combine different approaches.

Here's an example of how the structure skeleton might look like at the end.



You can combine different approaches in something more sophisticated, but it's hard to define the structure upfront.

I might be able to detect the structure relatively quickly. I've seen a few now, and business narratives start to look like horror or monster movies: the ones where you can tell the survivors after the first 5 minutes. But the structure *must* emerge from the team's hard work.

It has to happen step by step, so that everybody can master it. It has to be the result of a collective effort, so that everybody deserves a reward.

I am also skeptical of *"we already modeled every flow"* objections. More often than not the modeled flow and the actual one diverge. And a messy starting point leads to more interesting exception than modeling according to the "official version".

Highlight inconsistencies with Hot Spots

Enforcing global consistency won't happen without frictions. In fact we expect most of our inconsistencies and misunderstanding to surface during this phase.

This is when the role of the facilitator becomes crucial. While everybody is busy trying to make sense of the orange sticky notes, the facilitator should look for places where the discussion is getting hot and marking them with a purple sticky note.

[FIXME: picture]

Having these places - I call them *Hot Spots* - highlighted is super useful.

I like to leave HotSpots to the facilitator during this phase. An explicit call for problems too early in the workshop can create a flood of problems with a very low signal to noise ratio. We'll make hotspots accessible later in the problems and opportunities section.

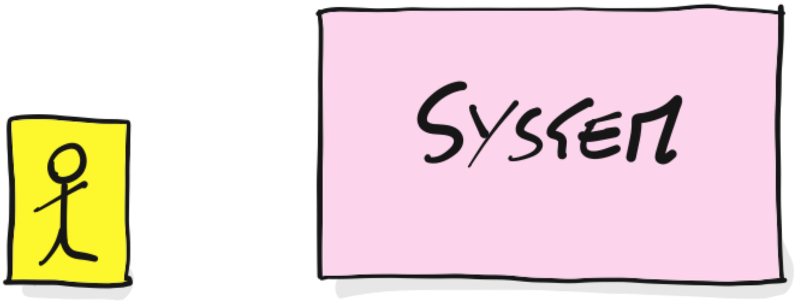
A safer target for finger pointing

HotSpots capture comments and remarks about issues in the narrative, and I am expecting to find quite a few of them. In fact, EventStorming provides a safer environment for going hard on the problem (which is now visible on the wall) while being soft on the people.

People and Systems

Events don't happen in a vacuum. We'll need to include *people* and *the external world* in our model, in order to better understand the existing forces and constraints.

We'll use yellow little stickies for people, and large pink stickies for external systems.



People and systems notation.

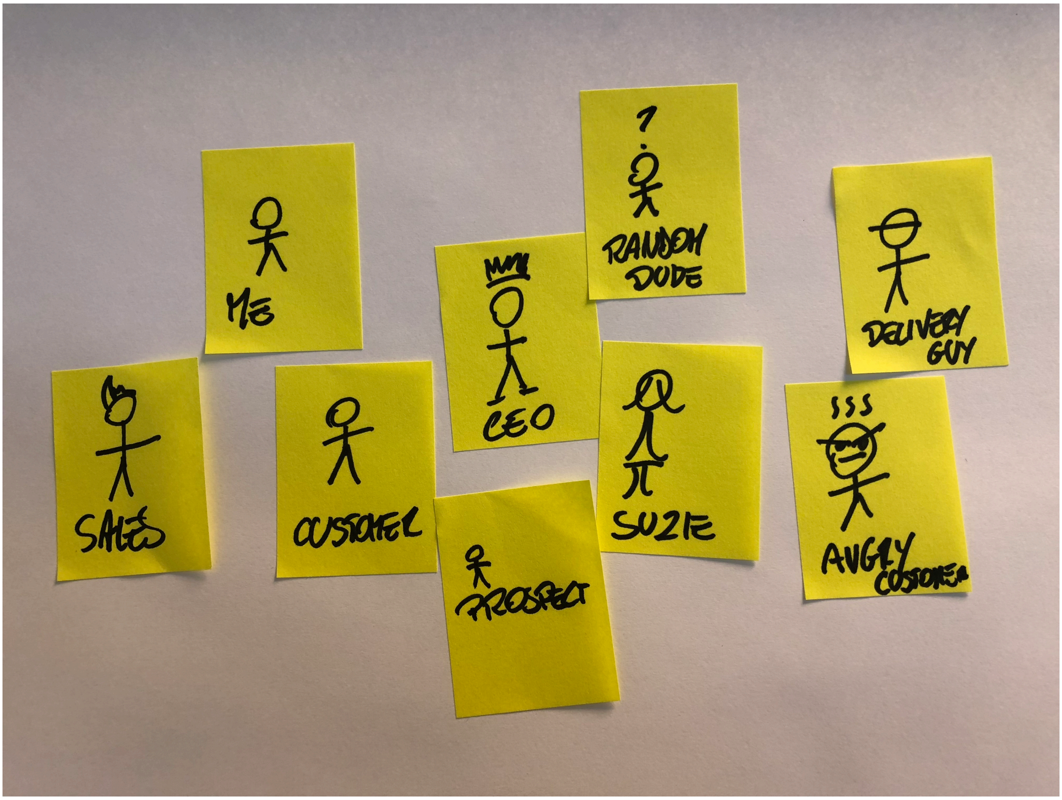
And we'll place them on the modeling surface, wherever they play a role in the event flow.

Fuzziness in action

I prefer to use the term *people* instead of *actors*, *users*, *roles* or *personas*, since it's not tied to any specific system modeling approach.

Fuzzy definitions allow every participant to be part of the discussion, without a specific background. Experts in a specific discipline like user experience design or business process modeling, might give away some precision in exchange for a more inclusive, background-neutral conversation.

This fuzziness will lead to a wide range of possible representations, from the most generic `User` to ultra-fine grained users like `John` and `Amy`, passing through all the possible variations of `New Customer` versus `Returning Customer` and so on.



Variations on the people notation are welcome!

The goal is not to match the right yellow sticky note to every event in the flow. Adding significant people adds more clarity, but the goal is to trigger some insightful conversation: where the behavior depends on a different type of user, where special actions need to be taken, and so on.

Don't be worried if these conversations are disrupting your current model. *This is a good thing!* It just means that you're starting to dig deeper, and learning more. Precision would progressively replace fuzziness.

External Systems

External Systems are the next big missing ingredient of our model. A business doesn't happen in a vacuum, and to see the whole system we need to see its moving parts, and the neighborhood too.

An external system is a piece of the whole flow which is outside of our control. Once more, we'll leverage fuzziness in order to trigger some reaction: some system can be internal (they're developed by our own organization) but external too (we're not the authors, just the maintainers, and we actually *hate* maintaining them), and this clarifications are usually really revealing.

An external system can also be something entirely different from software: a different department - *maybe the one that hasn't been invited to the workshop* - or an external organization.

In order not to make it formal - *and to make sure nothing will be excluded from our exploration* - I usually provide the following fuzzy definition for an external system.

"An External System is whatever we can put the blame on"

This may lead to some awkward, and apparently useless, sticky notes. A couple of times "Bad Luck" was modeled as an external system, we had "Europe", "Brexit" and "GDPR" too!

There's nothing wrong with that. In fact, that turned out quite useful: modeling `Bad Luck` prompted someone to write domain events for the unlucky events that could disrupt the flow, like `Missed Plane` in a conference organization. Some of the key dangers were now visible.

Whenever in doubt, and you feel there's something that might fit or not fit the picture, just put it on the wall. Something hard to categorize, like "the market" might be quite valuable to sparkle some interesting thinking. If it just adds noise, well ...you just wasted a sticky note. No big deal.

Is this an external system?

Fuzzy definitions will also push people into categorizing their own: *"Is this an actor or a system?"* can be an interesting question to ask (people might not care about the personality of the systems, a little more about humans)[13].

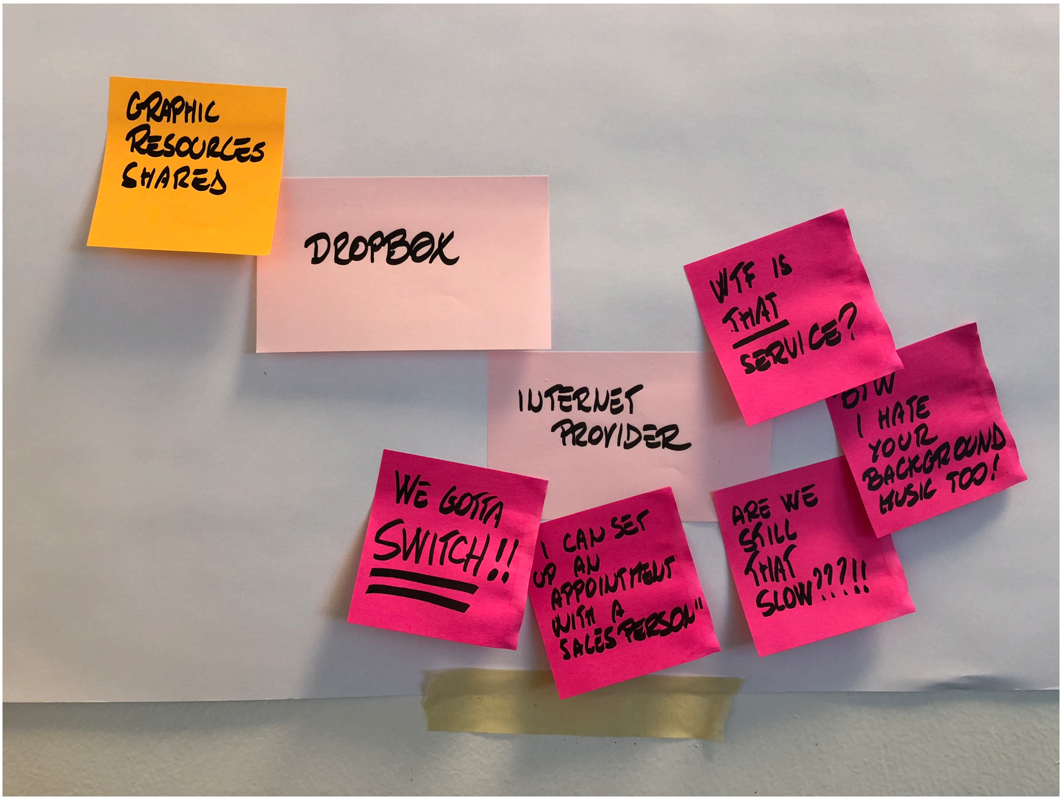
---

[13] In practice, I don't care whether a player in the system is categorized as a system or human, as long as it's visible. But I am really interested in the discussion.

It’s also funny to see developer’s behavior over a piece of legacy software: sometimes it’s *external*, sometimes this piece of software is *us*. Little language nuances will tell a lot about the real ownership, and the level of commitment or disengagement with software components.

One more target for safe finger pointing

Once systems become visible, it becomes hard to ignore them. Local cultural nuances and attitudes might influence this step, but I am usually alert for spontaneous comments (usually sarcastic complaints) that we should capture with *Hot Spots*.


GRAPHIC RESOURCES SHARED
DROPBOX
WTF IS THAT SERVICE?
BTW I HATE YOUR BACKGROUND MUSIC TOO!
INTERNET PROVIDER
WE GOTTA SWITCH!!
I CAN SET UP AN APPOINTMENT WITH A SALESPERSON"
ARE WE STILL THAT SLOW???!!

There will be no mercy for external systems.

Adding new systems, usually triggers the need for more events regarding things that may happen around those systems, like the need to renew licenses, or the mundane activities that happen on the boundaries.

With multiple systems involved, I prefer not to summarize too much: if space is running short is quite frequent to see people writing something like `Socials` instead of the more explicit `Facebook`, `Twitter`, `LinkedIn`, `YouTube` and so on, possibly hiding interesting details from the conversation.

The same goes with events, if this extra information is enriching your model, just add the corresponding events!

Phase: Explicit walk-through

Now, a more structured version of the flow is slowly emerging, probably still messy and blurry in a few places, but the whole picture doesn't feel solid yet.

A great way to enforce consistency during this phase is to ask someone to walk through the sequence of events, while telling the story that connects them.

When I say *'walk through'* I actually mean *literally*: talking about events while walking in front of the modeling surface will in fact trigger some modeler's superpowers.

Our voice will try to tell a consistent story: *"A user will visit our website, looking for special offers in the homepage…"* At the same time, your body will slowly try to walk forward, making you feel weird if the flow is not consistent yet and you have to move back and forth in order to touch the corresponding event.

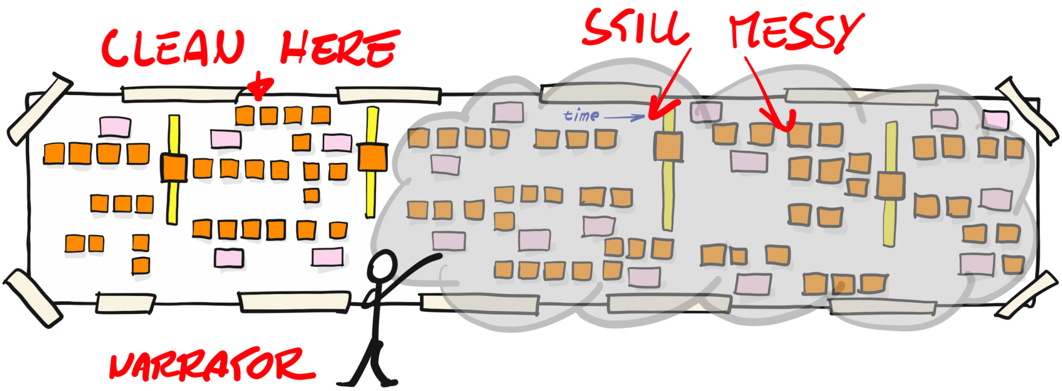
However, speaking out loud to tell the story will force your brain to think twice before saying anything, or to have some more thinking afterwards[14].

In practice, you'll be likely to experience extra thoughts like: *"Wait a minute! How will the user know about our website? Ouch, we forgot to consider the promotional campaigns and the Search Engine Optimization!"*

It is a good sign if your storytelling is bumpy and continuously forcing you to add more events. Your brain pain means that it's actually working.

---

[14] I have stolen the idea of talking out loud straight from Eric Evans and Domain-Driven Design: simply thinking about a sentence isn't enough to trigger the magic. But speaking loud, better if in front of some audience will.



The narrator "defragging" the overall flow.

It also means that it takes a lot of energy to be in the narrator's seat, because everybody in the room can (and *must*) interrupt you to challenge the current storytelling.

It is a good idea to change the narrator, in a relay race fashion, once we reach pivotal events *"and this is where Mary's team takes over"* offering the possibility to see the experts in action in their own territory.

While the narrator's on stage, the facilitator should make sure that the spoken storytelling is aligned with the model: missing events or systems can be added on the fly.

Manage discussions

Some discussions cannot be solved during the workshop. And narrowing the focus to one single issue might not be the best use of everybody else's time. When conversation is getting non-conclusive, I mark it with a Hot Spot (signaling that it won't be forgotten) and move on.

On the other hand, some discussions *are* interesting for everybody, and the workshop might be the one chance in a lifetime to get a long awaited clarification.

There's no default strategy on how to handle discussions on the spot. I like to observe participants' body language - it usually tells *a lot* ranging from

“Oh, please not again!” to *“Grab the popcorns, this is finally it!”* - and/or to ask explicitly what they would like to do.

It doesn’t make sense to kill a long awaited discussion just to follow the facilitator’s schedule. Big Picture is a *discovery workshop*: we can’t expect what surprises are going to look like, and we should be ready to discard parts of our plan in case we discover something more interesting.

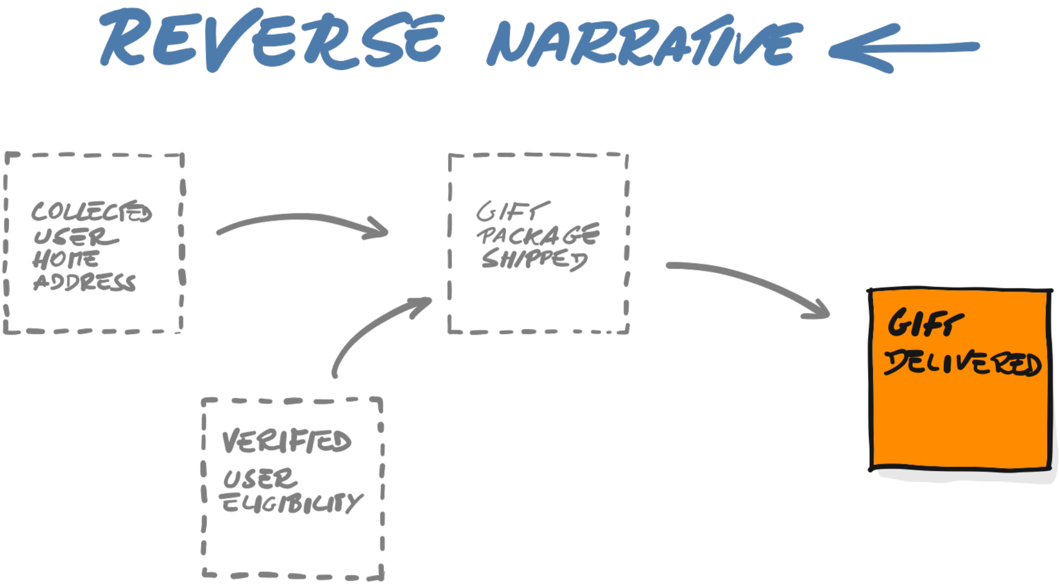
Reverse Narrative

Making sense of the whole flow merging the independent stories is probably the most demanding activity of the Big Picture workshop. Usually it needs to be rewarded with a break …because we aren’t finished yet.

The whole flow should now look meaningful and reasonably sound to participants. Their contribution tho the overall flow is now visible in the big picture. Unfortunately, this is still not the true story. The narrative that we just streamlined with the walk-through isn’t yet capturing the whole complexity.

It’s now time to challenge our model with a different way of thinking: a little trick called *Reverse Narrative*. Here is the idea:

1. Pick an event from the end of the flow, then look for the events that made it possible. The event *must* be consistent: it has to be the direct consequence of previous events with no magic gaps in between. Once again, speaking out loud will come in handy, exposing inconsistent assumptions.
2. If some step or piece of information is missing, we’ll need to add the corresponding event or sub-flow to our model.
3. Every other event needs to be consistent as well, so you may want to repeat it *ad libitum*.



Reverse narrative in action

You might want to challenge the audience asking something like *“So [Event A] is all it takes to have [Event B]?”* or *“What needs to happen, in order to have [Event C] happen?”*

Here is a little example from my company domain.

> Our company organizes public workshops and we’d like to provide attendance certificates to our students.
>
> Starting from the end, we bump into a `Certificate Delivered` event. Our paper certificates will be signed off by the trainer, but will need to be printed before the workshop. This calls for a `Certificate Signed by Trainer` and a `Certificates Printed`, but we ain’t finished yet: certificates are personal, and we need to acquire the attendee’s name before printing the certificate.
>
> *“We have them in the ticket information!”* somebody says, but it turns out that this isn’t always true: individual tickets provide this information, but group ticket purchased by large organizations may not be assigned to o participants, or can be reassigned at the very last minute.

> In practice, this means that we have a few more events: `Participant Name Acquired` or `Participant Registered` which is a direct consequence of `Ticket Sold` but can also be an independent event. Of course, we might also need a `Participant name changed` in case a company transfers the ticket to somebody else. Hmmm… I actually like the wording `Ticket Transfered` more[a].
>
> ---
>
> [a] The obsession with naming and rewriting is so Domain-Driven Design! You'll have a lot more of it in running a design-level EventStorming.

Reverse Narrative is a powerful tool to enforce system consistency. Even if we think we're done with forward exploration, we usually discover a relevant portion of the system (around 30-40%) that was buried under the optimistic thinking.

Both *Explicit Walk-though* and *Reverse Narrative* may take a lot of time. It's time well spent: it provides a lot more of insight and a feeling of solidity that was missing in the previous steps. But if you're under strict time-boxing constraints you might decide to sacrifice the depth of these phases, in order to complete the next steps.

Picking candidate events

Some events are natural candidates for backward exploration: *terminal events* (the ones at the end of the flow that seem to "settle everything") are a natural fit for this role.

I also like exploring from the *pivotal events* (the ones closing a specific phases) that often cover a checklist-like structure.

Bonus phase: Add the money

Whenever I run workshop with software developers, they invariably tend to neglect the money part of the flow: suppliers are not paid, invoices are not received or sent, and a large portion of the system that has to do with money is often overlooked or skipped.

I suspect that, for software developers, money lies somewhere in the intersection of *obvious* and *boring* problems. There's always something more exciting that having a look to the payment cycle, unless you are exploring the business flow of a *fintech* or cryptocurrency based company.

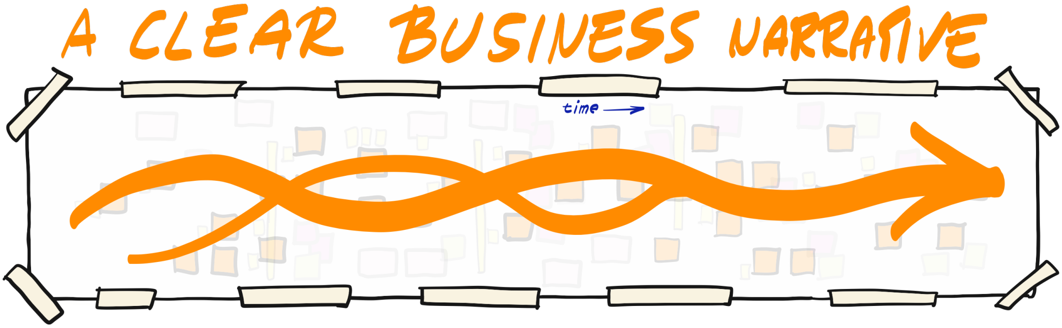
However, understanding money mechanics is vital for an organization's survival, and possibly even more for a startup striving to achieve sustainability. If the exploration looks naive, or if invitations didn't manage to bring the money persons in the room, it's a good idea to call a short exploration round focused on the money flow, in order to balance the blind spot.

To be fair, money is only one among the many different value currencies that could be victims of selective blindness, we'll talk more about it in playing with value chapter.

An emerging consistent narrative

Every step in our exploration brings clarity, details, and consistency to our understanding of the business dynamics. There's a lot of value in the emergent structure of the model: the shape of the map will stick in our brains for a long time, especially because we struggled to achieve it.

The exploration process ultimately led to interesting conversations. People that were only partially aware of the flow mechanics and constraints now finally see the whole thing, in the form of a consistent business narrative. The whole thing is all but perfect and shiny, but it should make sense in terms of storytelling.



A clear business narrative, delivering value to the key players in the system. This is what I am expecting to “see” at the end.

Well …that’s the intention! The effort on the storytelling should have highlighted also inconsistencies in the flow, which is now officially challenged and scrutinized.

Some steps might lack a clear motivation, look obsolete or unnatural. The larger the organization, the longer past solutions will survive even if the original problem they were supposed to solve is now gone forever.

Other steps will look like a pain, especially if the existing legacy processes and software are forcing customers and employees to extra activity whose reward is not in sight.

I can’t anticipate what you’re going to find, but you’ll know that you’ve found it when you’ll see that. Discovering inconsistencies might be embarrassing, sometimes. It’s better to focus on the opportunities that an exposed flaw in our storytelling might lead us to.

Phase: Problems and opportunities

Once we've included people and external systems, and we commented on them with sarcastic hot spots, we have reached a very interesting intermediate goal.

The whole system should now be visible

[FIXME: picture of a stretched landscape]

We might not be particularly interested in the mechanics of what happens inside the external systems, unless their internal mechanics are vital for our organization (imagine a company whose market is heavily regulated, they usually care a lot about what happens inside the regulator's organization).

What we mostly care about is what happens at the boundaries between us and the external systems. And in order to see the boundaries, or the no-man's land in between, a bit of "the other side" needs to be visible too.

This is also a great moment for spontaneous insights. Assuming that everything is visible, might also mean that somebody are still not seeing what they're expecting to see. In some workshops, this feeling might lead to an extra phase that scrutinizes value created and destroyed along the flow. This step is actually so important it deserved a chapter on its own.

Looking for key impediments

At this stage the grandiosity of our business flow, with constraints and impediments should now be visible in all its magnificence. It's time to choose what to do with that, but before choosing our next action it's better to provide everybody one last chance to have their opinion visible.

To do so, I now officially offer a 10-15 minutes time-box to add ***Problems*** and ***Opportunities*** to our model. Problems are represented with our familiar Hotspot notation, while opportunities will balance purple with a more optimistic *green*.

THIS IS A HOTSPOT: GOES FOR PROBLEMS, RISKS, ETC.
LIFE SUCKS!
WHY DON'T WE ADOPT BLOCKCHAIN?
THIS IS AN OPPORTUNITY: IT WILL CAPTURE IDEAS

Balancing problems and opportunities: a purple problem flood might seem overwhelming, but …we have a lot of green ideas to the rescue!

I expect most of the problems to have surfaced already, but this phase provides a safe chance to make your opinion visible without raising an explicit conflict, so it works very well in corporate scenarios or where interpersonal attitudes are an issue. I expect this phase to be relatively quick and quiet, but to offer some interesting surprises.

Green *opportunities* provide a perfect counterpart to the possible flood of problems. We have *solutions*, and problem solvers in the room too! There's hope!

Phase: Pick your problem

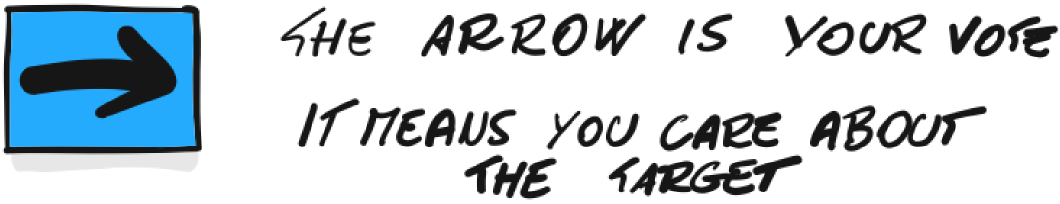
In a typical session you should now have a few dozens of problems and opportunities on the wall. Nobody will be able to solve everything quickly and nobody will be able to pursue every possible opportunity. So we'll have to find a way to guide our way out.

Time might be running short - this is the closing activity - but we might still need to quickly browse and eventually cluster the latest adding to the model before voting.

Arrow Voting

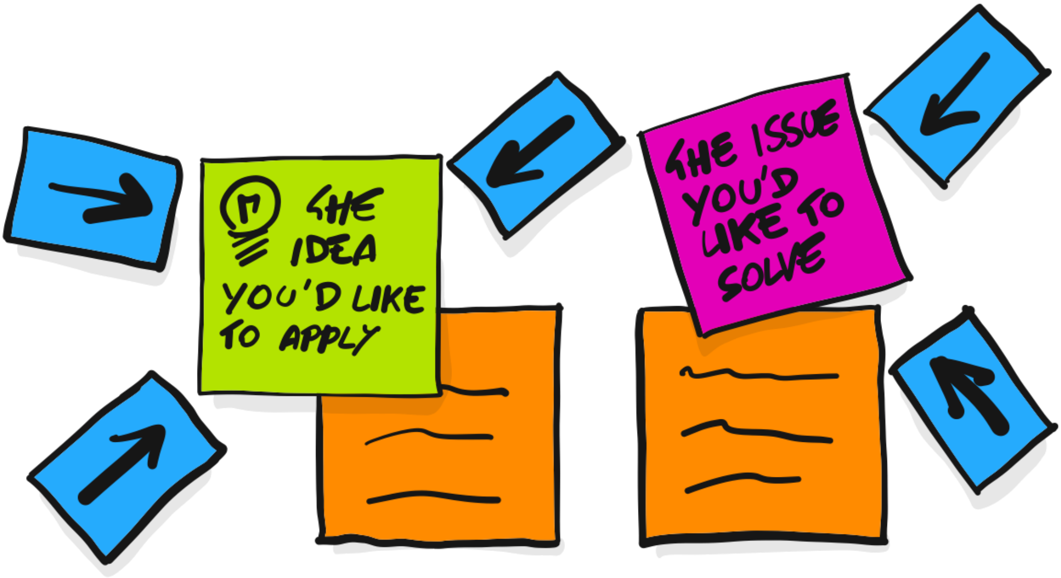
My favorite closing action for a Big Picture EventStorming is a round of Arrow Voting.

If you're familiar with agile retrospectives, you probably already know the practice of Dot Voting: arrow voting is very similar, *…just more expressive!*


THE ARROW IS YOUR VOTE
IT MEANS YOU CARE ABOUT THE TARGET

Arrow voting in action, pick your target!

1. Every workshop participants has the possibility to cast two votes. Votes are little blue stickies with an arrow.
2. Arrow should point towards a purple problem or a green opportunity. Everybody should be free to choose their voting criteria. I tend to use the words *"most important problem to solve"*, knowing that 'important' is a subjective term. Votes could be pointing to the same target or to two independent targets.
3. Voting happens more or less simultaneously, no voting catwalk.


THE IDEA YOU'D LIKE TO APPLY
THE ISSUE YOU'D LIKE TO SOLVE

Arrow voting in action, pick your target!

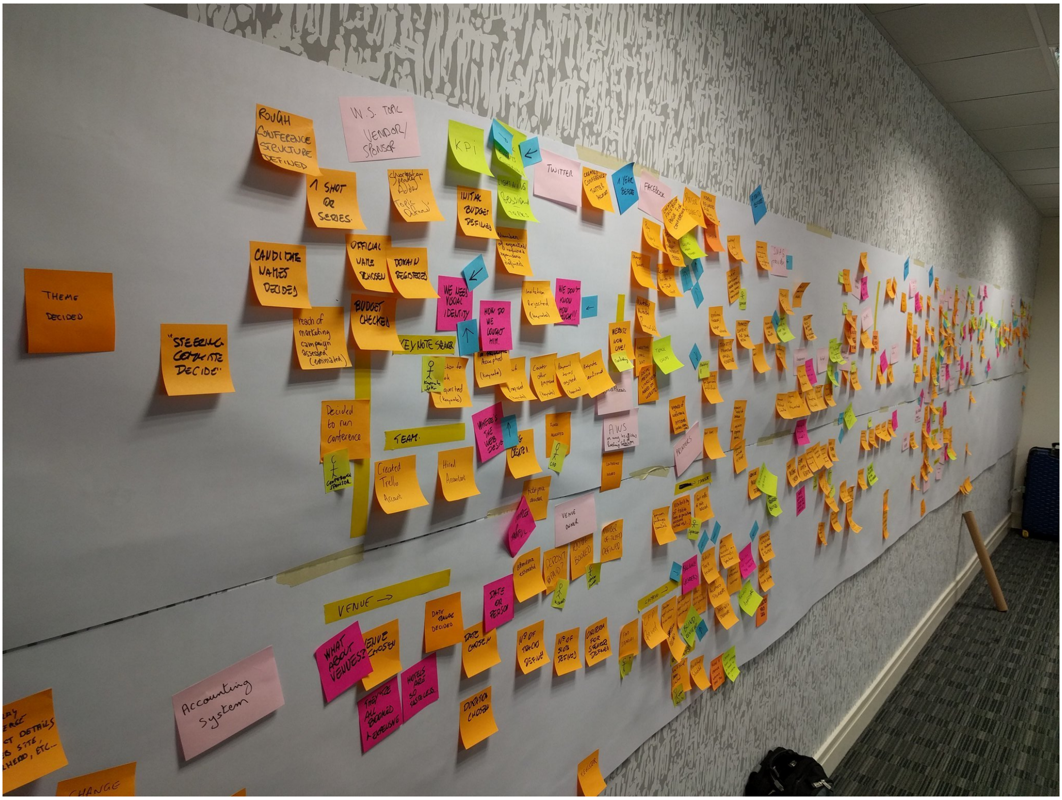
I am expecting a quick (usually a couple of minutes) round, where participants are again swarming on the modeling surface, choosing where to place their arrows. Body language can be again an interesting source of insights.

The facilitator should to make sure no power play is happening during this phase (especially in corporate scenarios), i.e. gently asking the main influencer in the room to wait a little to cast their vote, not to influence other participants.

Once people are detaching and getting in contemplation mode, it's the moment to wrap it up.

Seeing through the fog

After all this work (we started from a unsorted mess, we added some structure, we included people and external systems, we explored inconsistencies in a forward and then backward fashion) we should now have a clear indication of what to do on the next morning.

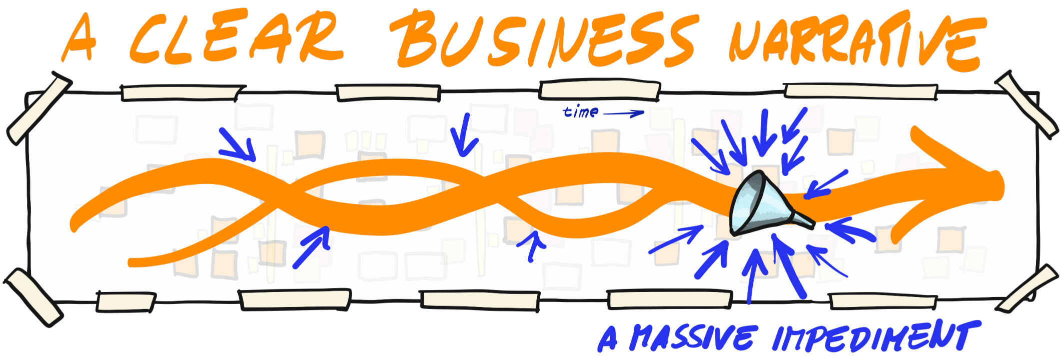


A picture of the outcome of a big picture workshop. More stickies than you ever imagined…

Apparently we now have a wall filled with colored sticky notes (a person just joining now, to *see the outcome* wouldn't understand much) but there's a little more to it.

We put together different expertises and perspectives, we achieved a deeper level of awareness about our organization structure, purpose and mechanics, and we voted (and highlighted with blue arrows) the most important issue to focus on.

Put in another way, this is what I am expecting to see through the fog.



A picture of the outcome of a big picture workshop. More stickies than you ever imagined…

This is the place where we should concentrate our efforts, the next morning.

Thank you guys! This has been a long day.

Should we always be voting?

Well …no! Voting should be triggered only if you can manage the outcome accordingly. Here are a few reasons why voting shouldn't be called:

1. Wrong people mix in the room: A single partisan perspective could be a trusted source of an opinion, but not a system wide diagnosis.
2. Wrong scope: exploring the whole was beyond our reach. We might vote, but the real key constraint might be hidden somewhere else.
3. Non-disclosure constraints: in a pre-sales requirements exploration scenario, an organization might be open to exploration, but not to explore their vulnerabilities.
4. Just too early: a startup in inception phase doesn't have a real impediments yet, just a list of assumptions to be challenged and questions to be answered.

In general, there are a few interesting variations over the Big Picture format, and context might drive us towards a different schedule. We'll go deeper on that topic in the Big Picture Variations chapter.

The promised structure summary

- **Invitations**: make sure you have the right people in the room, the ones who know and the ones who care.
- **Room Setup**: provide enough maneuvering space for your little crowd to work on the modeling surface in the smoothest possible way. Don't forget the basics: food, light and fresh air.
- **Kick-off**: make sure that everybody feels aligned with the workshop goals, possible warm-up round.
- **Chaotic Exploration**: everybody frantically starts adding the domain events they're aware of, to the modeling surface. Some interesting conversations can pop up, but people will mostly work in a quiet, massively parallel, mode.
- **Enforce the timeline**: let's make sense of the big model. Restricting the flow to a single timeline forces people to have the conversation they avoided till now. Structure will emerge from the common archetypes. More events will be added, but most of the work is *moving* events to achieve a meaningful structure. This is also the moment where *Hot Hpots* appear.
- **People and Systems**: which are the key *roles* in our business flow? Making people and systems evident, we are reasonably sure that if there are impediments to the flow, they are visible to everyone else in the room. This round is also another hot spot detonator: once everything is visible, people can't stop commenting.
- **Explicit Walk-through**: different narrators will take the lead in different portions of the system, describing the visible behavior and accepting the challenge of other participants.
- **Reverse Narrative**: if we're confident with the overall flow, we can ask people to think in reverse temporal order, or in *strict causal order* if you prefer. Quite a few events get added to the model, and the original flow becomes a lot more sophisticated than the original one.
- **Problems and Opportunities**: time to allow everyone in the room to state their opinion and ideas about the current flow.

- **Pick the right problem**: once everything is visible and marked with a hot spot, it may make sense to choose the most important problem(s) to solve. Sometimes you have clear consensus, sometimes you'll vote and have surprises.
- **Wrapping up**: take the final pictures, manage the closing conversations, do the needed cleanup, or postpone it if you can.

[FIXME: here a picture would be just perfect.]

☑ Chapter Goals:

- Know the basic mechanics of *Big Picture EventStorming* and the different phases.
- Understanding what can be achieved in a single workshop.
- Be ready for surprises that can happen during a massive discovery workshop.

5. Playing with value - part 1

Optional Step

Explore Value

Discover multiple currencies

Explore Purpose

What are you optimized for?

Chapter Goals:

- Discover how to explore value on top of an EventStorming session.
- Be prepared to ask and answer difficult questions.
- Be ready to choose what organization you want to be.

6. Making it happen

Managing Participant's experience

The facilitator is the primary responsible of the workshop user experience. His duty is to keep eyes open for impediments to optimal flow and remove them in the smoothest possible way

Postpone precision

The initial requirement (orange sticky notes plus verb at past tense) is apparently really simple. In practice, it's never met in the first round.

I am usually *strict* in enforcing the color scheme (if more colors are available, there'll always be someone picking a different color), but more relaxed in dealing with domain event in the wrong form.

The color problem is easily solved making sure different colors are not in sight. Wrong phrasing is a different matter. It will be an issue later, because it might hide some crucial complexity, but during the ignition phase is vital that participants don't feel under examination, especially in corporate scenario.

So I usually survey the model, and let it go initially, as long as wrong stickies are not the majority.

Unlimited Modeling Resources

The hidden cost of a depleted marker

You're just about to start a meeting. Ten busy colleagues join. After a quick welcome, you grab a marker and start writing the meeting goal on a whiteboard. Well … that was the plan. The marker is in its 'end-of-life'. Totally useless. You look for other markers… there's the red one, bit it isn't in better shape. Hrmpf. A volunteer leaves the room looking for more markers.

He's back in a couple of minutes, but in the meanwhile some vibe is already gone: you don't want to say anything important since one person is missing, but at the same time, people don't like just being idle. Some start a casual conversation, others quickly check their smartphones, in order to get some stuff done.

Your colleague comes back with some new markers, telling a story of the embarrassing questions he had to answer in order to get them. You try the new ones, just in case. Ten minutes behind schedule, you're ready to start, but one person is still engaged in the messaging thread that started while waiting. The room was booked for one hour, and you wasted 16% of that precious time. Counting 8 people in the room, this amounts to 1 hour 20 minutes of the average hourly cost. Way more than a marker. Even more annoyingly, the loss of focus and purpose can make the meeting even more ineffective. There's linear correlation between the time spent and the hourly cost, but the relationship with the generated value is more related to *focus* and *engagement* than the time spent. A disengaging setup could make the meeting pretty useless.

Visible Legend

Making it sure everybody knows what they're doing is crucial for the workshop's flow. Keep in mind that most of the workshop participants *are doing it for the first time*, so they'll need as much safety as you can provide. They're already far outside of the comfort zone, let's not make things harder than necessary.

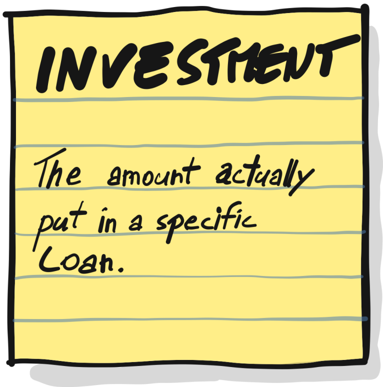
A **Visible Legend** where the current notation is visible and clear to everybody might come in handy to avoid impediments in the flow of thoughts (like *what did that pink thing mean?*).

It will also help a

Capture definitions

While everybody is busy adding events to the model, facilitators can leverage their own ability to ask stupid, oops …*obvious* questions.

If everybody in the room mentions a mysterious term an acronym, and you have the feeling that in the organization specific domain, this term means something really *precise*, that is obvious to everybody in the room except you, well …just ask for a definition and write it, then place it at the bottom of the flow.


INVESTMENT
The amount actually put in a specific Loan.

I use different stickies for the definitions, so that they don't get mixed with the core notation. In this case, everybody was talking about Investments, but the meaning wasn't obvious to everyone

More often than not, the facilitator isn't the only one person wondering what an Investment really is, but others may not feel confident enough to ask. If you get a silent nod of approval, you've done a good job.

Incremental Notation

Notation shortcomings

While trying to make sense of the whole thing, participants will soon realize that enforcing a timeline is tricky and probably simplistic. It does serve the main purpose of enforcing long awaited conversation, but it has some common shortcomings.

Can't we use Arrows?

Arrows would be just perfect in an EventStorming session. Unfortunately stickies can be replaced, and static pads can be shuffled, but arrows are *drawn*. This means that once you write something (and the paper roll is damn tempting) you can't easily erase that.

Not a big deal apparently. But it will quickly make your problem harder: the surface will look messier and dirtier, adding extra work for your brain in order to separate the signal from the noise.

More annoyingly, once an arrow is drawn, your brain will try to model the flow *without moving sticky notes* in order not to invalidate the existing arrows. This is a little instance of the *Sunken Cost Fallacy* that has the same dynamics: it will push you into more expensive mistakes in order not to admit the past ones[1].

I've found that people can get along quite well without arrows once a timeline is visually well represented. In the few cases where *proximity* and *temporal order* are not enough to imply consequential coupling between events (maybe because they're distant in the model) we can be more explicit and add a little note on the consequence event, or duplicate the originator event and place a copy close to the distant consequence in order to simplify readability.

[FIXME: this calls for an image]

---

[1] We'll discover that Sunken Cost Fallacy is our personal arch-enemy, whose influence spans from the lazy temptation not to rewrite a sticky note, once we find a better wording for it to the irrational affection towards blatantly flawed portions of legacy code.

Timeline is too strict

Time is the perfect baseline for narratives. But not every business flow fits easily into a strict sequence. Usually, during this phase, some recurring questions might be: *"How do we manage recurring events?"* or *"Where do we place events which aren't strictly related?"*

A timeline is a great way to force some degree of integrity between overlapping narratives, but strict compliance to the timeline is not our goal: it's just our best tool to enforce some consistency into the conversation.

However, here are a couple of modeling tricks that may be interesting for the picky ones.

Time-triggered events

Some events just happen, because it's time. Some financial closing operation might happen at the end of the month. Some more detailed report might be needed at the end of quarter, or end of year.

A time-triggered event, even if it's hard to envision it

If time is in the days scale, I like to use a calendar-day icon. One doesn't need to be an artist in order to draw it, but it delivers the message.

In general, when things happen 'at the right time' there's an implicit time-triggered event waiting to emerge. Introducing this concept as an explicit event could make the whole process a lot clearer.

[FIXME: another picture, maybe]

Recurring events

Some events might happen repeatedly. Even the just mentioned time-triggered ones may in fact occur at given intervals. If repetition is a relevant detail for a specific domain event we might want to annotate our domain event accordingly.

[FIXME: little picture of recurring event]

There is no main sequence

No business is just a timeline of things that happen in a strict sequence. There always will be feedback loops and things that happen at a moment disconnected from the main flow.

But the timeline is the best tool available to enforce consistency of the independent perspectives, and to trigger meaningful conversations. It makes the whole story consistent. And humans are very good in understanding stories.

The more I look into it, the more I realize that EventStorming is basically a support for *business relevant narratives*. As analysts or software developers, we should encourage domain experts to tell us *stories*, and ensure the building blocks of our model allow us to tell the same story, without inconsistencies.

Managing conflicts

A Big Picture EventStorming is not a workshop for those that like to sweep the rug under the carpet. In fact, its more of a *let's have a look under the carpet party*.

Chapter Goals:

- The role of the facilitator
- Tips for resolving conflicts
- Different people make a different party

7. Preparing the workshop - 30%

Choosing a suitable room

Too many times, I've seen the best intentions turned down by the wrong setup.

Unfortunately, you don't have to do much in order to have a wrong setup: it's usually already in place in your organization.

Your building was probably designed and built around the wrong stereotype of worker. Even if you had the privilege to design your office space around your needs, most of the times meeting rooms end up being optimized for a different purpose (which usually boils down to *"run ineffective meetings, so that there'll be the need for another meeting"*).

As Sun Tzu would put it:

> *Every battle is won before it's ever fought.*

You meeting room is your battleground. It doesn't provide you an advantaged position. In fact, usual room setup, people habits and inertia, will drive you to run just another long, boring, ineffective, exhausting meeting just like the dozens you had before. People will take a seat, and set up their laptop, starting to do something else while the somebody else runs the show.

You'll have to do something different this time.

It's your duty to *hack the battleground* in your favor, creating a special environment where individual and corporate habits won't get in the way. Your mission is to *find the right problem to solve*, not to *find the right problem to solve despite all the constraints of a hostile environment*.

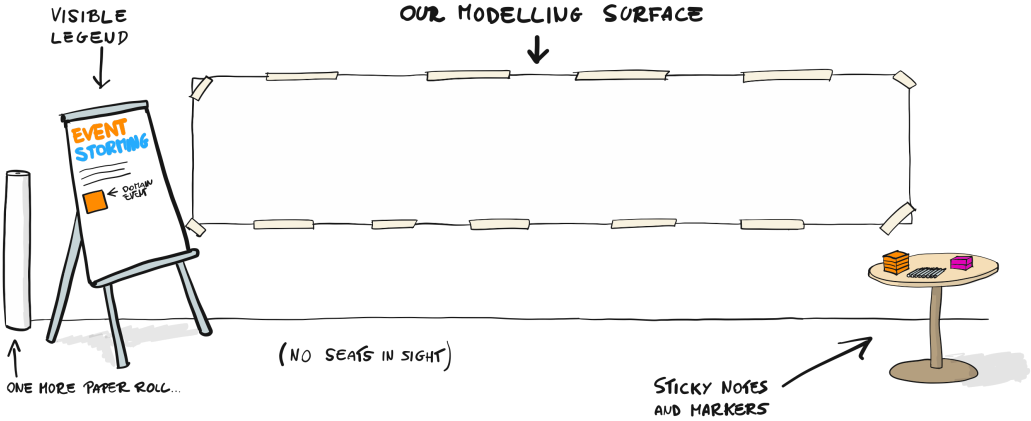
Instead of preparing yourself an alibi (*"we were given the wrong room to run the workshop"*), let's take control before the battle, like great generals do.

Here are a few key actions, that you have to consider, whether you're given a meeting room on your own premises, or if you have to rent a special one.

- **Provide an unlimited modeling surface**: usually this means placing as much as you can of your plotter paper roll to the longest straight wall. Be sure to have a strategy for expansion, the problem might turn bigger than you expected and what looked like *unlimited* in the beginning will become constraining later.
- **Provide a visible legend**: most of the people will be *new to EventStorming*, so be sure to always provide some guidance. You'll need a separate surface which is not interfering with your working surfaces. A flip-chart is usually fine.
- **Remove chairs**: You might want to have them available, later on, but definitely not at the beginning. Taking a seat invites a *passive listening mode* and a disengaged behavior[1] during the meeting, and having people stand up, after they took a seat is *really hard*.
- **Move tables aside**: Tables are normally a little harder to move away, especially those huge meeting rooms ones. We'll need some small table, in order to write on the sticky notes. Tall, smaller tables are a perfect match.

---

[1] *"I am sorry, I can't read from here…"*, it's so hard not to say a word, sometimes.



What the room is going to look like

Breaking Corporate Habit

In large corporations, people are so used to dysfunctional meetings that their Pavlovian reaction is to assume your workshop is going to be dysfunctional too. Around the meeting time (with all the typical cultural nuances[2]) participants will start to arrive, grab a seat, open the laptop and wait for the meeting to begin.

Don't let it happen! During these types of meeting the stakes are high and you'll need to use all the weapons available; and weapon number one is to turn the environment at your advantage.

In general, every time you hack the space before a workshop or a meeting, you have a sort of *'surprise effect advantage'*: many people coming to the meeting will notice there's something unusual and will be more curious about what's going to happen next[3].

This curiosity will be your best friend.

---

[2] the world famous *'Italian punctuality'*.

[3] to be honest, no guarantees here. You might as well bump into people whose attitude is *"Where the hell is my seat!?"*, but ...that's life.

Provide an unlimited modeling surface

In the previous chapter, we've talked about the effect of silos on the distribution of knowledge about key stakeholders. This ends up in problems which are hard to visualize on traditional supports like flip-charts and whiteboards.

EventStorming tackles this problem providing an unlimited modeling surface: a modeling space so big that it doesn't mandate any *scoping* before the action starts.

In practice, this often means the availability of paper rolls to be placed on a long straight wall.

[FIXME: image of the floor plan]

Too many times, our discussion has been constrained by the size of the available modeling surfaces. To discuss really big topics, the available space matters *a lot*. To overcome the problem, in my company every wall is now writable and finding a surface to model large problems is not an issue anymore. However, most working environment don't have that privilege, and calling a painter first, is probably not the best starting move[4].

[FIXME] More on this in the Breaking the space constraint chapter

100% Focus

An interesting side effect of the *no tables policy* is that there won't be space for laptops.

I am so sorry for that.

…No I am not. In fact I am just ensuring that people will be fully engaged in the workshop. It will happen anyway, EventStorming is way more interesting that checking corporate email, but there are also a few more dysfunctional effects of having laptops in the meeting that should be actively discouraged.

---

[4] we'll cover that topic in detail in the tools section.

Valuable conversations will be sparkling. An open laptop is anchoring a key person to be disengaged. Some conversation simply won't happen because somebody is busy discussing reimbursement of a taxi ride.

Some people might keep the laptop open as a reference, in order not to forget anything. While the professionalism is admirable, the result is not. The implicit communication will be something like *there's no need to waste time discussing, I've already done the exploration for you*. Which I consider to be a very annoying statement.

Maybe you don't need a meeting room

Sometimes, the problem of finding a *suitable room for an EventStorming* didn't need to be solved. A large enough corridor might just work as fine:

- no seats, *by design*;
- a long straight, wall;
- no table.

In this case you might want to provide a small table, for writing, but that's basically it. Moreover, not having a room provides a lot more visibility, allowing for somebody else to join the party.

Keep in mind that light and temperature also play a role, and that a stand-up only meeting can't last more than one 60-90 minutes without entering some form of *fatigue* state.

Keep the possibility of leaving stuff around

Choosing a temporary meeting room for running the workshop might not be the best idea, especially if you have to wrap up quickly to leave the room free for another meeting.

Even if the workshop is officially finished, there's still value in having the visible outcome available for a few days. Not every conversation or stream of thought that sparkled during the workshop could be time-boxed, and there will be invisible thinking that need just a safe place to land.

Moreover, the workshop is optimized for extroverted people, but introverted participants might have something very interesting to say, in a calmer setting, so prepare to leave stuff around and stay close to the modeling surface for a few hours after the workshop.

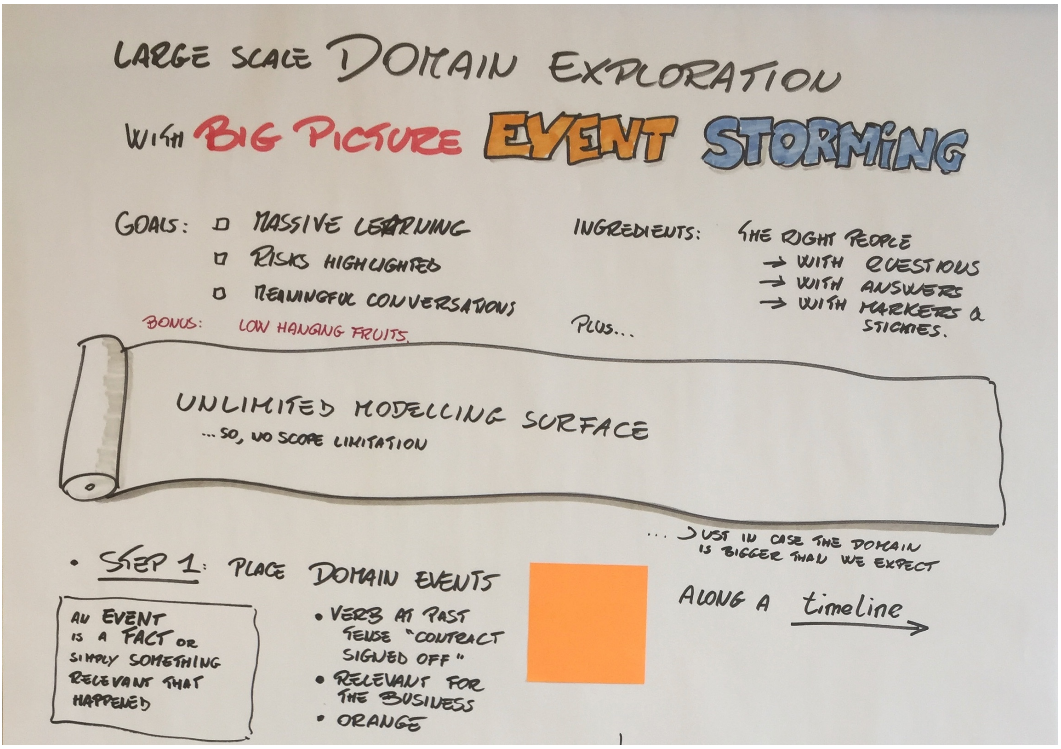
The morning after is usually really valuable, for afterthoughts. The ideal scenario would be to have the place available for one day and a half. But I said *'ideal'*, so trade-offs are part of the game. I'll tell more in the wrap-up section.

Provide a visible agenda

People are more inclined to join open-ended activities if they feel that facilitators are using their time wisely. Remember: the more important the people you're talking to, the more important will be the thing they're postponing because they joined your workshop.

A visible agenda is your best friend in order to make them feel comfortable, that someone knows what to do.

Here is a picture of a visible agenda from one of my workshops.



LARGE SCALE DOMAIN EXPLORATION

WITH BIG PICTURE EVENT STORMING

GOALS:
- ☐ MASSIVE LEARNING
- ☐ RISKS HIGHLIGHTED
- ☐ MEANINGFUL CONVERSATIONS

BONUS: LOW HANGING FRUITS.

INGREDIENTS: THE RIGHT PEOPLE
→ WITH QUESTIONS
→ WITH ANSWERS
→ WITH MARKERS & STICKIES.

PLUS...

UNLIMITED MODELLING SURFACE
...SO, NO SCOPE LIMITATION

... JUST IN CASE THE DOMAIN IS BIGGER THAN WE EXPECT

- STEP 1: PLACE DOMAIN EVENTS ALONG A timeline →

AN EVENT IS A FACT OR SIMPLY SOMETHING RELEVANT THAT HAPPENED

- VERB AT PAST TENSE "CONTRACT SIGNED OFF"
- RELEVANT FOR THE BUSINESS
- ORANGE

The starting agenda for a Big Picture EventStorming

I usually try to highlight the goals, even though I am aware of the intrinsic fuzziness. But the main goal is to provide an asynchronous reference available so that nobody gets stuck.

If the time constraints are strict, you may want to set an explicit plan for the different activities, and timebox them. However, keep in mind that Big Picture is a *discovery* activity, so surprised will override the plan. However, when this happens make it an explicit, and if necessary call a quick voting session.

Seats are poisonous

Managing invitations

There are three types of people you want to be part of your Big Picture EventStorming workshop:

- *people with **questions***,
- *people with **answers***,
- *a **facilitator***.

That's the *ideal* mix of people. But just like in parties and gigs, each one has its own alchemy. Let me clarify a little.

People with questions

Ideally, this description should include whoever is involved in designing and implementing a solution[5] to the problems we'll be discovering. For non trivial projects this involves a mix of organization, service design, software development, maybe product development.

Curiosity is the driver. The genuine desire to learn is the best reason to join the workshop. Whether you are the commander in chief and you want to grasp what is really happening under the hood, or you're the last person hired by the company but need to understand what's going on in order to contribute to the solution, you'll have plenty of good reasons to join the workshop.

---

[5] In the early days of EventStorming, when it looked like a great tool for speeding up software development, the term 'people with questions' seemed to match perfectly with the software development team. Later, EventStorming became a more general purpose tool that helped people coming from different backgrounds to have a better conversation around a shared artifact. Service designers, user experience designers, and whoever is involved in transforming the status quo into something different became welcome.

People with answers

Having all the experts in the right place and in the right moment is probably the nirvana of requirements gathering.

Gathering curious people is easy by definition, if they recognize an opportunity for massive learning, they'll show up like cats when you're opening the fridge. But when it comes to *domain experts*, those legendary creatures expected to share their wisdom with software developers in order to create great software solutions, well sometimes getting them on board might not be that easy. At least, not all of them at the same time.

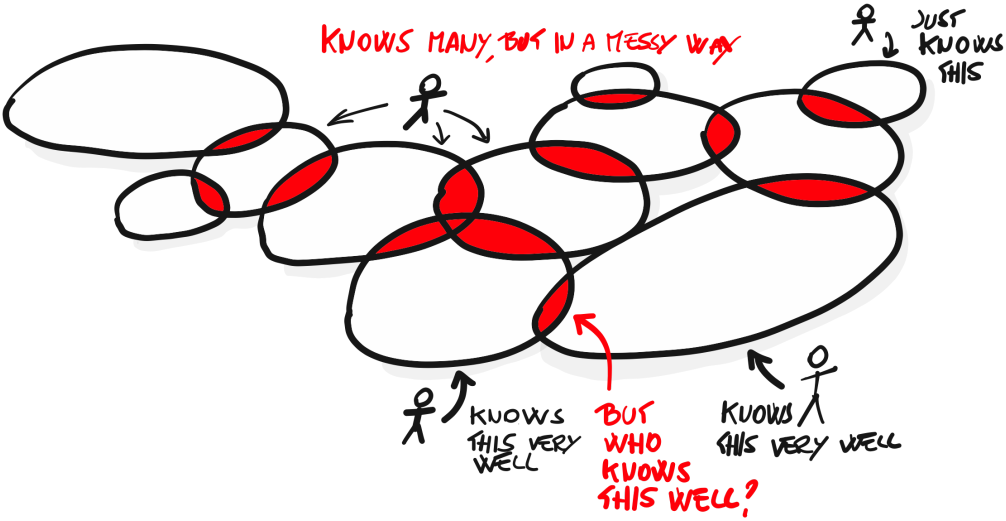
Well …invite them anyway. If they care, they'll show up.

People with *conflicting* answers

We already found in Chapter three that there won't be a single source of wisdom.

Let's recap.

> *There won't be a single domain expert, with deep knowledge of several aspects of the domain, but many people each one with partial, possibly outdated, conflicting views of the problem domain.*



Areas of expertise will overlap, but of course experts would not agree on broader definitions

The red areas, these overlapping, possibly conflicting, areas of expertise are *exactly what we expect to find*. A few people, with different stories to tell, each one telling a bit of truth, and discoveries to be made by asking the right questions to the right people. These are exactly the places that usually hide the best opportunities for improvements.

So don't be reluctant to invite people with strongly different opinions. Managing conflicts, clarifying them and making them visible, is one of the key outcomes of EventStorming, so … invite *that person too*.

You probably realized that I was trying to fool you

The distinction between people with questions and people with answers is very likely to be fake and artificial in practice. Nobody knows everything and intelligent people will be curious and have questions, no matter what! In a good workshop every single person involved will learn a lot.

But the workshop hasn't happened yet. People will be skeptical into investing their precious time into a weird activity, and the bigger their ego, the smaller the appeal of the words *collective learning* will be to their ears.

So the dichotomy *people with questions vs people with answers* only serves the purpose of extending the range of the people to invite, to everyone whose presence can be beneficial to the whole group.

The real story is that, at this moment we have no idea what "the real problem" is going to look like, so introducing early constraints on the solution shape wouldn't make much sense. Is it going to be more software? Or removing a piece of legacy software which is now in the way? Or just a different process and working agreement? We don't know yet.

Maybe, instead of the false dichotomy of people with questions and people with answers, it would be better to just invite *people who care*.

Facilitator

There's a lot of behind the scenes work that happen during an EventStorming workshop. And there's going to be some disagreement, that could lead to stalemates.

Since we'll be targeting problems lying under the carpet for ages, it's better to rely on somebody which is in the position to manage disagreements without much emotional burden, or a predefined position on the matter.

Facilitator duties will include workshop preparation, providing all the necessary stationery and the illusion that everything is under control.

Yes, I said illusion.

“If everything is under control you aren’t going fast enough”

Mario Andretti

Tech People or business people?

“Who should we invite? The technical people or the business people?”

Crossing hierarchies

I am not a big fan of hierarchies, especially if they are a roadblock towards a solution. EventStorming provides a common ground between people that see local solutions and people that need to see the big picture.

There is no need to make a classy distinction between *elite meetings* and *poor people’s meetings*. However, in many organizations, the chasm is there, and it won’t disappear easily.

The funny thing is that organizations where organizing a cross-department/cross-hierarchy is a pain *are exactly the companies that would benefit most from it*. But of course they won’t know it in advance.

Preparing the agenda

It’s very hard to convince people to join an uncontrolled meeting. So be ready to have something that looks like an agenda and set up a time-boxed structure for the different phases. But in practice, the facilitator will have to ignite a somewhat chaotic exploration with a very loose grip.

Conversation will sparkle. Some will be incredibly interesting: something the entire organization has been waiting for ages, and the facilitator will just have to *observe* and *take notes*. Others will be poisonous: the repetition of a disengaging ritual that will make somebody’s eyes roll, and the facilitator will have to find a way to stop or re-frame them, in order to keep the workshop valuable.

Invitations won't be perfect

There is no perfection in the invitation process. Or maybe I simply couldn't find it.

- The more people we invite, the harder it will be to find a suitable time to have all the key people in the same moment in the same place.
- The more we wait, the more we let the status quo rot.
- People will be skeptical about new weird stuff by definition. They will be pleased by the outcome later, but invitations are supposed to be done before the workshop[6].

Best scenarios I've been in were the ones where a sense of *purpose* was already in place. The whole organization was aware there was the need to do something, only they were confused about what to do.

But that's the workshop purpose.

The mess in the invitation is telling you something

If your organization is so complicated that putting together the key representatives of the independent silos is a daunting effort … *you have a clear metric of how much your organization structure is constraining learning*.

Is it a trade off you can live with? The answer is all yours.

[FIXME: Finish]

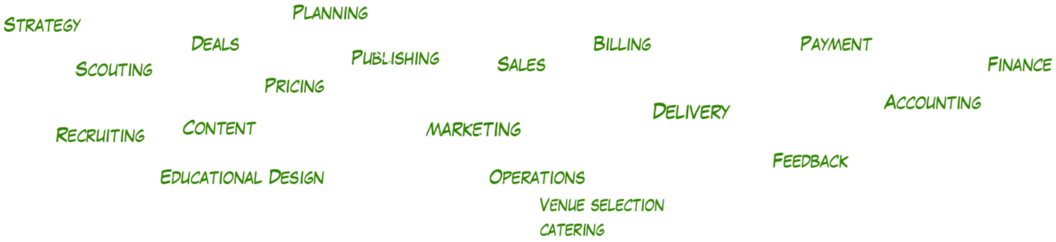
Manage expectations

---

[6] With the notable exception of Stephen Hawking's Time Travellers party, where the invitation was published after the party happened. No time travelers showed up, if this is proving anything.

8. Workshop Aftermath -20%

Cooperating domains



An emerging subdomains structure from my company's public trainings line of business.

When to stop?

I'd be happy to say something like: *"You can stop once you've achieved a satisfying level of understanding about the underlying domain"* or *"the workshop is over when every flow is completely modeled on the wall"*. Unfortunately, that's not the case: the dominant constraint for a big picture workshop is *key people availability*, so, unless you provide very high quality refreshments and beverages, the expected timebox is around two hours.

This re-frames our goal: we should maximize the value of the output, considering that we won't be able to finish the whole stuff. Our major concern would be to ensure we explored the more critical stuff in depth [FIXME: a clarifying picture here would be great], while not losing focus for the overall picture. Easier said than done, we'll dig deeper into this topic in [FIXME: link]

How do we know we did a good job?

Let me state it clear, the model we're building on the paper roll isn't the goal.

The model is basically two things:

- an *excuse* to trigger the right conversation with the right people,
- a *tool* to improve the quality of the emerging conversation.

Feelings

If an EventStorming goes really well, you won't need any checklist. A great session ends with people happily tired and a feeling of accomplishment.

You can actually catch yourself in a contemplating mood. Looking to the walls filled with colored sticky notes and a "there is nothing left to add" feeling. I like to call this moment:

"This mess is us!"

OMG it was awful!

If the feeling is different, it does not necessarily mean the workshop didn't work. A blind date isn't necessarily the beginning of a love story.

In fact some of the most effective EventStorming didn't deliver exactly what was expected.

In company X, getting the conversation to happen was extremely hard. The company owner acted as an Über-Domain Expert, and his subordinates didn't dare contradicting him. The workshop turned into a dysfunctional conversation with a silent audience.

Should I label this as a failure? Actually I was pretty happy: it took me less that two hours to have all the information needed in order to quit a project that was doomed. With so many personal issues involved, the possibility for a development team top deliver good software *regardless of the hostile environment* was *nil*.

In company Y, a key stakeholder interpreted the workshop in his own way. While everybody started placing domain events on a flow, he was putting his own stuff *on a separate portion of the modeling surface. The disconnection was striking. We managed to merge the two models together but it felt awkward at most.*

In this case, the problem was visible *before* the model. Body language and visible interactions showed us that there was a chasm between key stakeholders. Do you we think you can solve this problem writing good software? Good luck.

Even if the workshop outcome was somewhat bumpy, we had a clear idea about what to do next, that was *addressing the personal-political issue*. Starting the problem pretending everything was going to be all right, was not an option.

Visual Check

[FIXME: add an image with visual checks.]

Here's a little list of the things I visually check to be sure we dug deep enough.

1. Do we have **hot spots**? Did we have a good discussion about what looks like the most interesting problem to solve? Did we mark it as such with a purple sticky note? No conflicts and no problems doesn't mean honeymoon: it means somebody was missing, or maybe *lying*.
2. How many **Domain Events** emerged during the discussion? For a two hours workshop 100 to 200 is probably a reasonable number. Less than one hundred is telling that you only scratched the surface. A special case might be if you're working with startups.
3. Did you capture all the **External Systems**? They're usually sources of variability and trouble. If they're not displayed on the modeling surface, then you haven't been exploring large enough.
4. Did you check for *"what is missing from this picture?"* explicitly? Without an explicit offering for it, people might just skip some vital details, just because they might not look relevant.

Wrapping up a big picture workshop

Even if it looks like the party is over, people are leaving and it's you and a couple of colleagues cleaning a room full of markers and

Book the room a little longer

In some companies, reserving a slot in meeting room could be a messy task. If you're in one of such companies, and quitting is not an option[1], be sure you reserved enough time for the cleanup. There's nothing more devastating than another group of people knocking at your door, telling "We've reserved the room", while you're in the heat of the discussion or in thoughtful contemplation. You'll need time to *physically* wrap up, to cool down and analyze ideas, and you'll need to do it *in the room*. A good EventStorming usually triggers a lot of conversations and insights, and not each one of them has a proper closing during the workshop. Body language information also cannot be annotated during the workshop, but double checking the signals with a colleague at the end is very important, not to miss or misinterpret the implicit messages.

What should we do with the modeling artifact?

[FIXME: Image, the secret move to fold EventStorming]

Keep it around for a few days

If you have the possibility to keep the artifact, just do it. There are a few good reasons for leaving the big thing around for a few days.

- It provides a visible reference for people that didn't join the workshop. Those that weren't invited, or that declined the invitation because it sounded *just like another meeting*.

---

[1] Think about it, if you end up postponing problem resolutions due to the scarcity of adequate meeting rooms, you have a problem. If you're lowering down the [FIXME: finish the note.]

- It provides a visual anchoring for the ones that joined the workshop. Given the amount of information processed in a large workshop, it is very likely that some afterthoughts will happen away from the office. Having the model in place the morning after will help capturing and eventually discuss them.
- It may trigger new reasonings or sparkle new conversation: the model is intended to be *readable*. If more people want to join the conversation, that's usually a good thing.

Archive it

Manage psychology

[FIXME: weight the chapter, it could be included somewhere else.]

Let's start from the big picture model, the one that encompasses the whole business flow without getting deep into the design artifacts.

Managing the big picture artifact

The real outcome of an EventStorming session is the *cooperative learning*. Even if getting finally to the big picture felt exhausting, the real outcome is that *you've been there, had the discussion and created the model*. Don't fall too much in love with it: *the model is still wrong*. The workshop environment makes it easy for participants to spot mistakes in the exiting model - leveraging the wisdom of the crowd - but some inconsistencies could only be spotted by coding and testing the model in the real world, possibly with real users.

Focusing on the hot spot

Sometimes, the exploration of the model draws the attention towards some hot spots. Long lasting problems are usually the topic of some colorful conversation. If there's consensus around a hot spot (on the problem location, finding the right solution is a completely different matter) there should be no excuses, let's start exploring the problem a little more and work on it. The worst thing one can do is to spot the problem, gather consensus on it and then lose all the momentum by *doing something else instead*. This approach comes straight from *Theory of Constraints*: once you spot the bottleneck in your system, this is the area where every little improvements counts.

Usually, I tend to keep the model around for some time (a week, maybe) to leave some space for afterthoughts and possible refinements of the model. But once the big picture view is captured, the value of the visible model decreases, and there might be better uses for the square meters of visible surface. Most of the time, we take a picture and fold the roll. This makes us feel safer, because we're not loosing our work, but the real story is that we seldom look back to the model.

However, you have to consider that I developed this ascetic detachment from material artifact after several sessions. Your workshop participants are nowhere close to that, so *don't force that detachment* by thrashing the model. It will take some time for them to realize they actually didn't need it any more.

So, again, take pictures for digital use (a panorama for the whole flow, and some on-the-spot for better readability), roll the paper again preserving all the stick notes, and store the paper roll in a safe place, to be used just in case.

Multiple hot spots

If many hot spots have been marked on the model, it's probably a good idea to keep the roll around for some time, in a "Ok, what's next?" fashion. Once hot spots are removed or displaced by new implementations of the process, the team might look to the next bottleneck in line or to the new one that just appeared. The roll becomes a nice place to have a periodic discussion about the intermediate results achieved in an effort to improve organization performance. Of course, the roll won't be enough: spatial and colorful representation is a good thing, but real measured data is more than welcome.

Prioritizing hot spots

No hot spots

Having troubles is not always necessary. If no particular pain points are highlighted on the roll, then just capture the knowledge about the process structure. Now the main goal is not in removing visible impediments but in catching opportunities like implementing some new feature or adding some variation in the existing process.

Documenting the outcomes - TRICKY

Emerging structure

What are we looking for?

How do we know that we're heading in the right direction? We'll see later in *When to stop?* that our workshop will probably be limited by external factors such as people and room availability, so we won't have the privilege to decide when the workshop is over.

However, we'll need a way to understand if our discovery is heading in the right direction: baking a cake might be easier if you already have a good idea about how the finished one is going to look like.

Multiple Models

- Models should be **internally consistent** in terms of language

If you have a software development background you'll probably recognize two familiar concepts from *Domain-Driven Design*:

- **Ubiquitous Language**

- **Bounded Contexts**

However, the spirit of the workshop is not to introduce new concepts to domain experts, so, *please*, resist the temptation to start explaining Domain-Driven Design to your boss and just *use it* to your advantage.

The bare minimum boils down to 2 concepts.

- *Language is revealing*: stay alert for *differences* and *synonyms*. Things are often not *exactly* the same.
- *The whole thing won't be consistent, only small portions will*. Stop trying to transform it.

9. Big Picture Variations - 50%

[FIXME: filler]

Software Project Discovery

A variation of the big picture format can be used when exploring the landscape for a new software project. This is the typical scenario for a software/consulting company engaged to develop custom software for a different organization.

In this scenario the setting can never be perfect: the software company would probably try to invite all the key people in the customer organization organization, in order to learn as much as possible from the workshop, but the key people wouldn't probably be so easily available, since the true value of the workshop would be visible only in retrospective. Political issues and power play have a role too, but the trickiest problem has to do with the customer organization's maturity: if they're used to a waterfall approach to software development project, or simply strict budgeting and scope definition, it will be hard to convince them of the need for a workshop instead of the traditional specification document.

Sometimes the best approach is *to not even mention EventStorming*, but simply bringing stickies and paper roll with you and run a small scale workshop, *"just because we're used to see things in this way"*. This way you may skip the effort of convincing somebody to to something they don't know anything about, and possibly can surprise and engage them.

Or you may want to call an explicit workshop as the gateway condition to pick the project: it's a big risk reduction effort on both sides, but the perception around it may vary a lot given the relative market placement of the two parties involve.

In general, invitations are not a perfect science, so be prepared to trade-offs, and to get the best of what is reachable, while seeding the ground for a *perfect workshop* next time. When a workshop is embedded inside the pre-sales negotiation cycle, it will invariably suffer of all the dysfunctions of negotiation in the complex domain[1].

Core problem or just Supporting?

In order to have the biggest impact, the software company would probably challenge the customer organization in order to better understand the problem they're facing and the overall scope.

As we've seen [FIXME: precise reference], not all problems are born equals: some can be game changers[FIXME: make sure the concept is clarified before], some mere gateway/enablers. The software company ultimate interest might be to look for the best fit given their strengths: some projects require special people (possibly with an explorer/pioneer mindset) and can boost revenues, others will just be mandatory gateways in order to stay in a given market, but they'll be mostly budget driven.

Unspoken conflicting goals

Many times, the project scope to be explored is just a portion of a biggere scenario:

Organization Retrospective

Induction for new hires

A few companies started using EventStorming as a way to quickly bring new hires up to speed. I am doing the same in my own company, and works amazingly well.

---

[1] My friend and colleague Jacopo Romei explored the space around knowledge work negotiation deeply in his book Extreme Contracts. [FIXME: Links]

However, there's a few meaningful differences from the standard format, which are worth mentioning.

1. Can't invite the whole team *again*. If the company is on a hiring frenzy with new people joining the company on a weekly basis, you can't simply re-run a Big Picture workshop. Good news are: *"If you already ran a Big Picture workshop, then every participant already has a better understanding of the whole"* so being a proxy expert in a downsized workshop isn't much of a risk.
2. It still does make sense to re-discover the whole thing. Just showing the outcome of a previous session won't work very well: for the explainer it will be a summary and a good reminder of conversations that happened during the workshop, but for the newcomer that wasn't there, there won't be any memory to attach to.

The recommendation here is to run the workshop, giving the leading role to the newcomers, starting to model the system based on their guessing and assumptions, and progressively correcting them. The way we do it is to ask "Let's start modeling what you think it's happening in this organization!" and see where it leads. There shouldn't be no expectation of correctness - people are too fresh to know everything - but having their wrong assumptions visible is a great place to start from. The senior member of the team will have to provide explanation to justify the correction, and together they should evolve the model accordingly.

Why is it working?

It took me a little to make it work. It took me ages to understand why it was working.

10. What software development really is - 40%

Few people have done more harm to software development as a profession than those advocating that “software development is like building a house”.

Software development is writing code

This basically stating the obvious: *everybody knows that software development is writing code*.

That’s the reason why we use frameworks to write less code and speed up software development, and we deliver in days what used to take months.

That’s the reason why senior developers are more efficient than junior developers: they type faster and know more keyboard shortcuts, making the more productive.

That’s the reason why pair programming is a scam: if only a person in two is typing, software will be delivered at half speed.

That’s the reason why adding more people to a project guarantees earlier delivery. The more people involved the better. Type like the wind, dudes!

That’s also the reason software development is *so predictable*. We only need to estimate the lines of code to write to have a high-quality / high-reliability projection of the delivery date.

By now, the level of sarcasm in my writing should have reached disturbing levels. Every experienced software developer knows that the above sentences are crap. Unfortunately the story isn’t that simple.

Developers aren’t the only people thinking about software development

Every other development in your company is thinking about software development in this way. Probably including your boss. Your customers are thinking about software development in this way too. Chances are that you’re thinking in this way too, when thinking at someone else’s software or when presented with a new cool code generation tool.

Even if you reached a level of consciousness that makes you avoid that thinking trap, there will always be somebody around you thinking in this way.

Sometimes software development *is* just typing

Unfortunately, you can spend ages convincing your boss that software development is something more complex than just typing, and finally one day you bump into one of this task which is just plan labor. Not repeatable enough to justify a script, but dumb enough to provide somewhat linear estimates: *“it took me 2 hours to clean up that page, it will probably take me 3 days to clean up the remaining 15 pages”*.

What many managers don’t get is that this is just an *exception* to the normal flow. Software development isn’t repeatable. When it’s repeatable, we can replace it with a script. But linear projections look so computable. They look so *clean* in an Excel spreadsheet (sarcasm level rising again).

Software development is learning

The real story is that software developers are spending a relevant amount of their time *learning*, in order to do things the don’t have a clue about. This is not only related to a technology or a programming language[1], but also to problem domain, context and so on. Differently from other professions, we’re doing things for the first time, most of the time (even if it looks like the same old typing from the outside).

---

[1] When was the last time you thought you knew everything about your programming language? For me it happened only in 1983 working with ZX Spectrum’s Basic language. Because all the keywords were visible.

In practice, this means that we can achieve the unicorn-like state of flow mostly when we know enough of the domain to proceed solving problems one by one without getting stuck (assuming that we are somehow free from interruptions).

But it also means that many times, when we embark on a development task *we have no damn clue about what the solution is going to look like*

The sweet spot of programming

[FIXME: something] Think about that: if the problem hasn't been solved, then it's worth writing some lines of code. If it's not

> When we're looking more productive, it's probably the moment we're not adding that much value[2].

Can you estimate learning?

Seriously, can you? Take a book, for example. Yep, take the beloved *Domain-Driven Design, tackling complexity at the heart of software* and answer these two questions (assuming you haven't read it): 1. Can you estimate the time needed in order to *read* it? 2. Can you estimate the time needed in order to *understand* it? The first answer is relatively easy. A simple projection of a mechanical task: reading a dozen of pages every evening before sleeping will make you through the book in a reasonable time. But what about the second one?

[FIXME: somewhere in bold working code is a side effect]

---

[2] Many years ago, in a project full of folkloristic behavior, there was a developer that had really long, silent and mysterious coding activities. However looked like he was in a state of flow, most of the time. Taking a closer look I discovered he was adding *getters* and *setters* manually to DTOs and domain classes. When I asked him why he wasn't using the IDE built-in code generator, he said: *"I prefer this way. It gives me peace of mind."*

The unconfessable truth of institutional learning

The whole idea that *coding* is actually learning, and that real world programming is actually a hunt for a moving target is totally missing from institutional learning. Coding exercises are defined against a well-defined assignment, which is clearly written, and doesn't change along the way. I have the feeling this approach might be e

Self taught programming anyone

Across the years I've come to recognize a different breed of programmers: those guys that felt in love with coding and moved way beyond the expectations. It can happen in many ways. A pet project is the classical form. As a teenage programmer I had started a few video-games. But even in university assignments I often ended up adding more stuff to the coding exercise, just because it was cool. However, in pet projects, you are both the programmer and the stakeholder. You change your mind, you'll need to make trade-offs. And both roles learn along the way.

This doesn't happen in institutional learning, and we're swimming in the consequences.

Software development is making decisions

This is a tricky one, and probably the one which is making the software development profession so close to *creative writing*. We're taking decisions all the time. We're taking *trivial* like how to name a class, a method or a variable (which can anyway have severe consequences in the long term), but we're also taking *nontrivial decisions* like how to implement a given solution, which library, component, product or even programming language to choose for the next project, up to the spectrum of

Oh come on, everybody is making decisions!

Software development is waiting

Yes, that's every developer's dirty little secret. We wait. We wait a lot. We wait for the compiler to finish the build. We wait for the customer to send us a clarification. We wait for the test suite to complete the test run. We wait for the sysadmins to grant us permission to pass through a firewall. We wait for the meeting room to become available for a conference call, or a modeling session. We wait wor a new supply of markers and stickies. We wait for the boss feedback the could destroy a month worth of efforts.

We wait. A lot. Every day.

But that's not the worst part.

The worst part is that we're so ashamed or bored of waiting that, while we're waiting for something that *we start doing something else in the meanwhile*.

Chapter Goals:

- Our mental model about software development is mostly wrong
- We've been trying to optimize the wrong thing
- Have a look at Kanban
- Have a look at Theory of Constraints

11. Learning is the problem, so what? - 20%

Understanding our ignorance

The moment I realized learning was the key constraint in my profession I felt betrayed. I spent years in schools and universities on a learning path that felt useless and that totally ignored the real nature of the problem: if software development is a learning process, I - as a professional software developer - had the duty to be (or become) a *professional learner*, but there was nothing in my career path that led me there. Or so I thought.

Learning is not studying

The moment I put learning at the center of things, I realised that many colleagues had this mental shortcut: *learning equals studying*

How does learning happen?

Pressure Kills learning

The Dreyfuss model as a constraint for learning

[FIXME: reference to Kevlin Henney's talk at BuildStuff 2014]

The invisible deliverable

It’s time to talk about the elephant in the room. Every time I talk with software developers about software development as learning I consistently get approval.

I am aware of the biases that are tied to my position: i said *“software developers”* but in reality my sample consist mostly of *software developers who are into Domain-Driven Design* or more generally who are hanging around in the same communities where I do. But that’s not the point.

The real problem is that

[FIXME: Finish that]

The documentation lifecycle

Learning without action is waste

The argument for learning has some drawbacks. People tend to associate learning with the vituperate *analysis phase* of the waterfall software development process. [FIXME: I’m talking about it in *A deep dive into problem space*]

That’s definitely not the case for EventStorming

The bottleneck is a call to action

The other reason why you definitely shouldn’t let the outcome of a Big Picture EventStorming session fade away is in the nature of the exploration.

Theory of Constraints quick recap

In a production system, there is always **one constraint** which is limiting the system throughput. Improving the performance on the system constraint - the *bottleneck* in ToC lingo, improves the performance of the whole system.

Prove your learning with some coding

Learning

A real iterative process

☑ Chapter Goals

- Learning cannot be pushed
- Look for impediments to learning
- Still, some time and errors will be needed in order for learning to happen

People Interaction

“It’s like a finger pointing away to the moon. Don’t concentrate on the finger or you will miss all that heavenly glory.”

Bruce Lee

12. People In Action - 25%

There is something that scares me in the words *"put all the relevant people in the same room"*. It is definitely not an *idea*. I wish I had invented something, but this is definitely not the case. Putting the key people in the same room is plain *obviousness*.

Still, there are reasons why people won't do that.

I guess I've probably done something more similar to saying *"the emperor is naked"*. If we play devil's advocate with the idea of having all the key people in the same room, we can end up with an interesting list of objections:

- Will be wasting people time: more people means more hourly cost (linear) and more organization cost (close to exponential), so we need a decent return on time spent to justify the effort.
- We've been in meetings like that. They're dysfunctional, endless, boring ceremonies of stating things that we already know.
- If we're lucky we can witness one more episode of 'corporate ego clash'

[FIXME: complete]

Managing Invitations

It shouldn't be a problem to invite the right people in a startup. The company is still small enough so that gathering everybody in the same room, might be the default setting.

In larger corporations, even reaching the right people might be a problem. A key factor is the level of sponsorship you have inside the organization. As a consultant, I often have a privileged position, since the level of sponsorship is visible from the negotiation phase. But if you are already inside the organization, it is very hard to subvert the rules and call a special workshop involving people that are incredibly busy, expensive, or so high in the hierarchy that you only know about them by reading financial newspapers.

“How do I convince my boss?” is a losing approach. Convincing somebody to take a risk for a new thing that it’s *damn hard to explain before experiencing* might be a dead end alley.

I usually recommend to start small, having a time-boxed and scope limited workshop (maybe just about a few key features) deliver some value, while discovering some hidden impediments that might be in the way. You won’t challenge the foundations of your project, or your company strategy, but you’ll gather enough information to do a better job with your project, and a better leverage if you’re planning to scale up to a Big Picture EventStorming.

Igniting the workshop

Group Conformity

Every group will show some degrees of conformity. If everybody else in a given context (a room, for us) is behaving in a given way, we’ll feel the pressure to adhere the standard. Even when the group decision is clearly wrong, a vast majority would still choose conformity over making a stand for their own belief.

If you want to freak out about this, take a look to some footage[1] of the experiment conducted by Solomon Asch: it would probably make you change the opinion about the human race.

Signaling loop

What happens in your brain, happens also in the brain of other people in the same room. Unfortunately, we’re not completely aware of the signals we’re sending.

---

[1] https://www.youtube.com/watch?v=qA-gbpt7Ts8

The problem with the fire alarm

It turns out that if there's a fire in the building you're in, you have a higher probability to survive if you're alone in a room at the moment the fire alarm starts.

The reason? Once again, *conformity*. The sound of the fire alarm puts a person in an unusual situation. The typical reaction is to look for someone else, to understand what is the right thing to do. If you're alone in your room, you'll probably follow your own judgment, and quickly leave your building (probably with your laptop).

But if you're running a meeting with some colleagues, then you'll be looking to the behavior of the other people in the room in order to get some clue. Unfortunately, in the absence of a real expert, other people will be doing exactly the same thing, looking at you. The *"let's wait and see what others are doing"* gets misinterpreted for *"this is not really dangerous"* and you might stay some minutes more.

There's a pretty disturbing video capturing this experiment too.

Hardly reversible actions

In order to successfully manage the workshop we have to be aware of the role played by these forces. They'll be there anyway.

Taking a seat: by taking a seat, you're signaling to everyone else that is now ok to take a seat. Once you take yours, others will quickly follow. However, now the chaos is gone: instead of acting as a swarm in front of the modeling surface we now will be more inclined to act like *spectators* looking what others are doing, especially if we're sitting at the other side of the room. **Standing up**: standing up and adding one more sticky note shouldn't be that hard, but once everyone is sat, then *standing up* and placing a hot spot purple note becomes close to a riot act. It takes a lot more of energy especially if your note can be read as a criticism to someone else in the room.

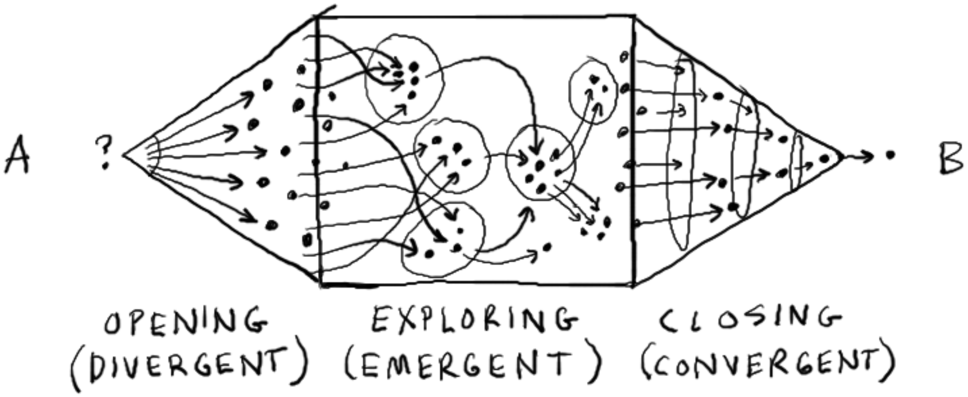
In general, once people are sitting it is very hard to have them standing up again, unless you're offering free food and beverages or some compelling reason to stand up, like *going personal* for example.

of these always present forces will play

Mastering Workshop Dynamics

The GameStorming Approach

The best source of information for managing workshop dynamics is Dave Gray's book GameStorming, which proposes a simple and powerful model for collaborative activities.



The canonical phases of a workshop according to GameStorming (courtesy of Dave Gray / @davegray @xplane)

- In the **Opening** phase, we try to maximize *divergent thinking*: we'll need independent, contributions and we'll manage workshop dynamics accordingly.
- In the **Exploring** phase, we discuss and mix individual independent ideas, often triggering new insights and *emerging* thoughts.
- In the **Closing** phase, we're converging on a certain output, often selecting the most important insights and discarding some of the weaker thoughts.

In practice, the facilitator has two very powerful weapons that allow him to manage workshop execution: the room setup, and the rules of the game.

Mastering parallelism

When it comes to EventStorming we have to be aware that there are different forces in play, partially contrasting each other: *

Observing Body Language

This is my land

Here is the problem

IT'S NOT EXACTLY LIKE THAT...
A PHYSICAL MODEL PROVIDES A SAFE SUPPORT FOR SPOTTING CRITICAL AREAS IN YOUR BUSINESS PROCESS.

Visible inconsistencies usually trigger immediate reactions

Attacking the problem not the person

Let me state it again: providing a detailed physical representation of the domain allows problem to be highlighted and tackled without turning the issue into a personality clash. When facing a problem, or a threat, our brains are wired to look for a concrete representation of a problem. If this is missing then we start looking for the second best: normally the person *reporting*[2] the problem or the one which for some reasons is associated to the problem. Unfortunately, if we react aggressively to the problem we often end up attacking the person[3] instead, a process issue becomes a nasty personal issue, with all the unintended and unnecessary emotional overload. In order to avoid the issue, some people might be reluctant to speak, to avoid blaming anyone. But avoiding to discuss of critical issues has never been a winning strategy.

A red sticky note on the wall with the name of the issue is a good and cheap target for blame and criticism, instead of a colleague. Sticky notes don't get offended or overreact to blame. Critical issues can be discussed with a lower risk of getting personal.

Sometimes people just want to go personal

Pursuing political correctness at all costs, might not always be viable. We are investigating the problem but we don't know it. I'm really happy whenever I can see a person problem as a system problem and attack the system instead, but there

[FIXME: reference to negotiation]

The magic of finding all

I have no clue

I don't belong here

Proximity and distance

A workshop can be a great chance to see people dynamics in action. Physical proximity can be a great indicator of some weird dynamics going on

---

[2] There are quite a few traces of this effect in common wisdom, such as *don't shoot the messenger* or *ambasciator non porta pena*.

[3] Yes, I am ranting. Sorry about that.

Enforcing narratives

Nobody likes chaos

The less surprising discovery was that, even without explicit guidance, teams were autonomously steering towards some form of layout organization for their domain events.

- Some teams naturally followed a ***left-to-right temporal order***: the aforementioned *timeline*.
- Some were clustering events according to some ***domain-specific affinity*** (i.e. sales related events separated from delivery events and so on).
- Some others clustered events according to the relevant ***actors***, in a more user-centered fashion.
- Despite our efforts to avoid providing specific guidance, not a single team was in pure chaotic brainstorming mode.
- Well… this should say something about how thick is our safety net when we try to get into freestyle modeling. There's always somebody adding some structure to raw data.

Different approaches led to different insights faster, but also hid some of the potential of the timeline approach.

Why a timeline is better

However, the most striking evidence was that not following a timeline was somehow making things more difficult than necessary.

Sequential exploration is more natural

"Customer adds an item to the cart, and then checks out, proceeding to payment" or *"when an order is placed, then the corresponding packaging instructions are sent to the warehouse"*. When exploring business flows we normally chain

events in a roughly sequential order. *'Then'* is the magic word, linking some events to their followers.

Enforcing a timeline allows people to *see the sequence* and *spot gaps* between events: *"There should be an 'Order Payment Accepted' between a 'Cart Checked Out' and an 'Order Shipped'!"*. Inconsistencies are more visible (it's easier for us to spot gaps in a narrative than in a categorization) and this promotes a more robust *wisdom of the crowds* effect.

Enforcing narratives

[FIXME: here or in People in Action?] I realized that one of my unconscious sources of inspiration came from the TV show 'LOST'. Watching some special content on DVD I was fascinated by the way the screenplay was laid down with stickies, keeping track of all the different parallel stories and relevant events. The authors didn't explain how they wrote stories, but what I saw as a background for their interview was already inspiring.

Later, seeing this brilliant Movie Narrative Charts from XKCD[4] probably completed the job. My brain was hooked.

The thing is …*we are not collecting requirements!* We're having a meaningful conversation! And in conversations, *people like to tell stories*.

Stories are the way knowledge is spread among people. *"Driving and drinking is dangerous"* is lame compared to *"Bob was driving home after a couple of beers that night. Hit another car, and two girls were injured. Now he's facing a trial and seriously risking jail."*

Stories are how myths and religions were spread among the non literate.

Stories are what hooks people on TV, teenagers on the phone, coworkers at the water-cooler and old friends together at the pub.

Stories are what people likes to tell, and also what people are probably ready to listen.

Stories are also *consistent* in some way. People capture cause and effect relationships between facts. Well people actively look for causal relationships even when there are not.

---

[4] https://xkcd.com/657/

Walking through scenarios

The apparently innocent water-cooler question

There are two recurring traits in the personality mix of an EventStorming workshops: * Developers are focusing on very unlikely worst case scenario * Domain experts are painting a rosy too simple scenario, in order not top confuse developers.

[FIXME: finish!]

Walking Skeleton - redefined

The term ‘Walking Skeleton’ has been made popular by Steve Freeman and Nat Pryce in their book Growing Object-Oriented Systems, guided by tests and refers to [FIXME - quote the original definition], but the original definition comes from Alistair Cockburn[5]:

> A Walking Skeleton is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.

Is it really a failure?

In some cases, the output of the

---

[5] You can read the whole thing on Alistair’s Wiki: http://alistair.cockburn.us/Walking+skeleton

Nobody likes to be wrong

Brainstorming biases

There have been quite a few discussions lately on the real effectiveness on Brainstorming sessions.

[FIXME: link]

EventStorming is not Brainstorming

Despite the *_storming_ in the name, EventStorming has little to do with the original intent of Brainstorming session.

A Big Picture workshop is essentially a *discovery* workshop: there isn't much creativity involved and different people are supposed to contribute with different perspectives on the same storyline. However, the risk of group thinking producing inferior results could still be present.

It is a specific facilitator's duty to look for signals of power play and conformity that could reduce the possible spectrum of information. And to manage it accordingly (see also single out the alpha male in the patterns section).

Chapter Goals

- The Ubiquitous Body Language
- People matter more than the model
- The Walking Skeleton, redefined
- Workshop success criteria

13. facilitating the workshop - 10%

In the previous chapters we hinted that an EventStorming workshop is like organizing a party. Here is where we explain how to be a great Disk Jockey.

The economics of a meeting with the big guys

At the risk of being boring, here is some basic maths:

How expensive is a meeting?

Well …this also helps us to define or goal [FIXME: there is duplication around this one]:

> *The goal is to make sure that the workshop will be more valuable than the overall time invested.*

What about missed opportunities?

There are many sinister consequences of thinking only in terms of *costs*. A more precise way of thinking is in terms the missed opportunities. It's not the hourly cost, it's *"what I could be achieving instead"*.

Time is not linear anyway

However, while the individual participation cost can be calculated, the return of investment is completely non-linear [FIXME: reference to the learning chapter] and heavily limited by physiology and energy cycles. Once people are tired, it doesn't make more sense to insist, *just to put all the required stickies* *on the wall*. If people are tired, maybe the best thing to do is call a timeout and go out for a walk, the wall took all of our attention, maybe some interesting conversation might happen where the wall is not dominating.

Promote questions

Allow people to checkout

After the initial kick off, it might be clear that some of the participants are not playing a key role in the incoming discussion. There are plenty of reasons not to be fully engaged, and if somebody thinks he doesn't belong

Allow people to stay

The reverse is valid too. Sometimes the meeting has already a well defined agenda: "We are going to run a big picture exploration in the morning, and a smaller meeting focusing on the technical details for feature X in the afternoon". I happened to have the one of the best conversations in the second one, with a non-technical domain expert, that asked to stay longer, just out of curiosity[1].

Keep your mouth shut

Introverted and extroverted

---

[1] Well, to be honest the simple sound of the word *curiosity* puts me in a good mood. Probably because it's the best indicator that you're doing a good job in promoting learning.

14. Breaking the space constraint - 30%

> What happens in our brain when we're provided unlimited modeling space? What happens when we don't have that privilege?

What makes EventStorming so different from whiteboard modeling? Here are a few reasons.

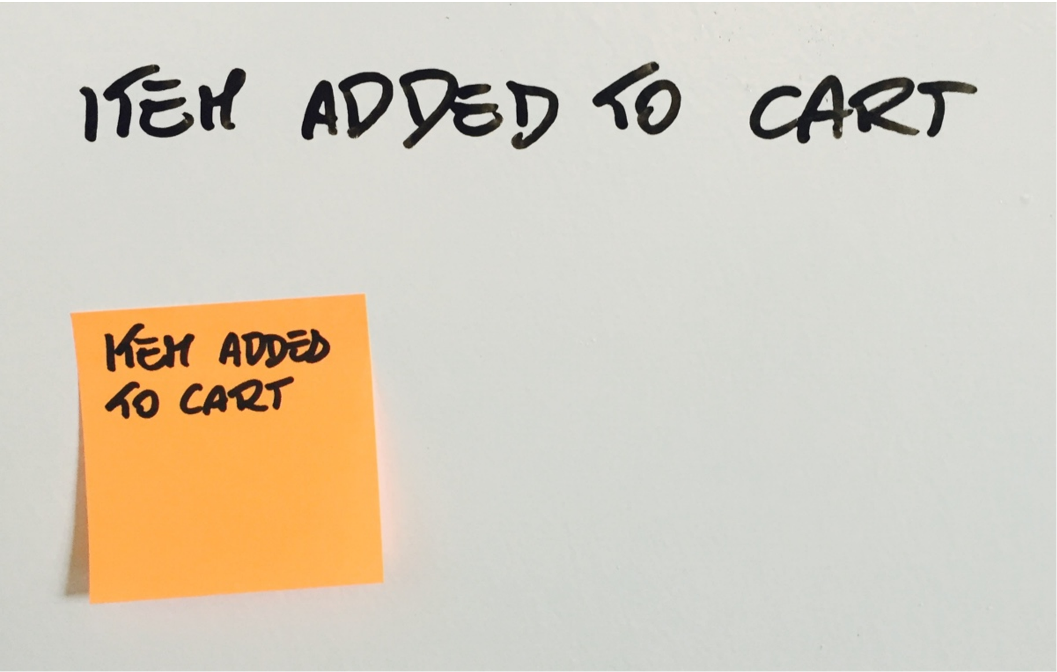
- Using a paper roll or a generic unlimited modeling surface, allows the team to *model more complicated topics*.
- Using sticky notes allows for *more items in the same amount of space*.
- Using colors allows to *better understand the structure*, promoting emerging *repeatable modeling patterns*.
- Having more space and more markers allows for *parallel modeling* in different portions of the available space.
- Using stickies makes rewriting and relocating cheaper.
- More space and more people allows for natural grouping of people themes and ideas.

Overcoming your whiteboard implicit constraint

Font size matters

There's another little trick in the way we can model using sticky notes: we're using the space available in a more efficient way.

Consider the amount of space required to write a class name like `ItemAddedToCart` in handwriting on your whiteboard, compare it with the space needed to write the same thing on a sticky note, and we're done.



It's so embarrassingly obvious…

A whiteboard tip is usually fatter. No way we can write that small and be readable. Writing on whiteboard takes around three times the space needed to write the same text on a sticky note.

Hold on! You used two lines! Are you calling me a cheater? Yes I am! Stickies force us to use a smaller space, whiteboard doesn't, or does it in a more annoying way.

Sunken cost fallacy

Regardless how good you are, your model is going to be wrong.

My handwriting sucks

Everybody thinks that good handwriting is something like a *gift* and there's nothing one can do about it. The real story is different: handwriting becomes better the more you practice it. So here is my little tip: Get a good marker. Write to be readable. Don't you like the result? No problem. Relax. Throw away the sticky note and write another one.

The illusion of the unlimited model

The ideal modeling surface is infinite in extension and height. Problem: we can walk horizontally, but we can't fly, so an infinite height modeling surface won't help.

Is your roll width enough?

For the first exploration round, a single roll would be enough. Later you'll probably hit the point where a single roll won't be enough, here are a few symptoms. - contributions to the model will slow down; - people will tend to frown more often, and the problem apparently will look harder to model; - people will ask to put more clarity or to separate the flow into swim-lanes;

Do you have enough orange stickies?

A secret rule of hotels, when managing buffets, is to have more food available to reduce food waste. When facing a rich buffet, people are taking just what they need to eat (the hungry brain is pleased to see there's enough food for a second round), while with less food offered, people tend to overfill their dishes, worried that some specialties won't be available for the next round.

The brain mechanics with orange stickies are no different: try to have as many orange stickies available as possible. When only a few of them are available, your brain might be signaling something like *“if I start modeling so finely, then we’ll run out of stickies”* that will turn into a not so interesting macro-level modelling[1].

Isn’t the whole thing against the principles of *Elements of UML Style*?

Yes, one of the key points in Scott Ambler’s book was that one should not draw more than seven boxes on a diagram, in order to be understood. Here the problem is different: we’re modeling together, and when exploring large systems, the constraint doesn’t make sense. However, when digging into a specific portion of the system, my suggestion is to remove the clutter and start exploring on a new surface.

Chapter Goals:

- what happens in our brain when we don’t have enough space?
- how to understand that uncomfortable feeling in our brain?
- How to achieve unlimited space
- How to master the mechanics of unlimited space?

The Lettering Kata

Here is something one can do to improve his lettering skills.

---

[1] Thanks to Massimo Azzolini for making me aware of this one.

Modeling processes and services

In this section we'll explore how a EventStorming could turn from a discovery tool into a collaborative design tool, promoting interdisciplinary collaboration between different stakeholders, in order to design meaningful services, features and products.

15. A platform for cooperation

Process modeling is the scenario that benefits more from interdisciplinary collaboration. In order to model a satisfactory model, it is important that all the key stakeholders are involved: *business stakeholders*, *service designers*, *user experience designers*, *software developers* just to name a few.

16. Exploring value

What is the ultimate purpose of the process you are designing? Or of software you are writing? Which way are you going to make your users happy?

These questions are often overlooked in software development projects. The feature is in the backlog, so there is clearly some business value attached to it. I mean …product owner [FIXME: complete]

Once you have a clear representation of the whole business flow, you might want to put it under scrutiny, by testing the human factor.

This can be tricky, because everyone in the organization is supposed to know about their own business, and asking questions where everybody *obviously* knows the answer doesn't look like the most productive use of people's time.

To put in another way: this is the type of apparently silly questions external facilitators are more entitled to ask than internal employees.

Explore Value Created and Destroyed

Look for the main transaction

Business processes usually happen around a specific transaction. In fact, many of them exist exclusively only to make this transaction possible.

> My company Avanscoperta[a] offers public training classes for advanced professions. In such a scenario, the main transaction happens when the training is delivered and attendees learn for the things they paid for. Except that …*it doesn't work that way!*
>
> A naive implementation of such a process would be that people pay at

the door to join the training and generate *knowledge* by learning. But that wouldn't work in practice. *Tickets* need to be sold in advance in order not to put the day at risk, and allow for more robust planning. So some value is created by early sales in terms of reducing *uncertainty* around a public training date. Discounted Early Bird tickets are a common way to reduce this uncertainty.

The main transaction is then split into parts: *money* gets transferred to the organizer the moment the ticket is sold (well …depending on the chosen payment type), while knowledge is hopefully delivered to the attendees during the training class. The distance between the two key moments in time opens the door for all the nastiest cancellation and refund combinations.

From the trainer perspective, value is created and destroyed all along the way. Some trainers might find valuable for their *reputation* to find themselves in good company on the company website, others might find annoying to be asked to provide picture and bio. When it comes to planning, trainers manage uncertainty too: a late cancellation can become a problem, because there might be no time to find another revenue source for the now empty day.

---

[a] http://www.avanscoperta.it

What if the key transaction is not in scope?

In software systems, it is relatively rare to have the real transaction happening inside the software. In Amazon, customers place *orders* on the website, but goods are delivered in the physical world. Kindle reader might represent an exception, but while possession of a paper book might originate value by itself[1] the kindle version doesn't work that way. Value is generated only when the customer actually *reads* the book.

---

[1] if you're that kind of person that tries to make an impression by making pictures of yourself, with your huge bookshelf in the background.

Explore multiple currencies

What to do when there is no value?

Look for inconsistencies

Chapter Goals:

- Processes are not a given thing, they must prove valid.
- Money is not the only value to look for.
- Sometimes, value creation is just behind the corner.

17. Observing global state - 10%

> Banks are so worried about eventual consistency. But every time I transfer money I see money disappearing in a wormhole and reappearing somewhere else a couple of days later.

The transaction obsession

Every business transaction can be seen as a process of reconciliation, filling a gap between current and desired state.

I might be feeling hungry while walking by John's Street Food van. If the smell is good and I have money in my wallet, this situation might just trigger a business transaction - me buying a hot dog, maybe - and the necessary actions to satisfy the need.

However, this transaction doesn't happen *atomically* like database transactions would. I won't handle a 5 € banknote with my left hand, releasing it to John only after I have managed to eat my hot dog with my right hand only, possibly in a single bite.

In fact, I will just ask for a hot dog, and pay for it, leaving John the possibility of running away and start a new life with my shiny 5 €, without serving me the hot dog I just paid for. For about one minute I will experience the thrilling possibility, of losing my money. Or maybe not, because 5 € is not tempting enough and, assuming John is a reasonable person, it wouldn't make more sense to run away with my banknote hoping to never meet me again[1].

---

[1] It turns out that most of our social conventions rely on not having a strategy for dealing with guilt. We're assuming there are rules, and playing by the rules makes sense. It would be extremely embarrassing for John to meet us again a week later, after running away with our 5 euros, *unless* John has some kind of strategy like: *"What the hell do you want from me now?"* that would usually make the righteous one completely freak out. If you're getting good with this, then you'll loose friends, but …politics is your career.

A Transaction is just a process

If we zoom into business transactions, we'll discover that they're never *atomic* but they're rather a sequence of states which are somewhat inconsistent. Let's explore more in detail a slightly different scenario, where I take a seat, and pay my hot dog before leaving.

1. I am hungry, there's money in my wallet, John's is open and the hot dog smell is in the air. The price list is visible, I can afford the hot dog.
2. I say hi to John, now he knows I might want to order something. He won't ignore the next part of the conversation.
3. I order a hot dog, and a bottle of water. Now John knows what I want, and I know that John knows it.
4. John hands me a bottle of water and tells me to get a seat. Now I have a bottle I haven't yet paid for, but I don't have my hot dog.
5. I hear John yelling: *"One hot dog!"* before talking to the next customer. I am now confident somebody is cooking my hot dog.
6. I open the water bottle and drink a little. This is clearly a non-reversible action. I won't be able to return the bottle, now. Or maybe I will but I am just too educated to try.
7. A couple of minute pass, and I start getting impatient. You don't mess with me when I am hungry, and 2 minutes is more than sufficient to prepare a hot dog.
8. My hot dog has been prepared, but I can't see it from my table. Even if I look nervously towards John. This is a very unfortunate configuration, because every second lost here will cool down my hot dog (decreasing its value), and increase my impatience. Making me wait a lot for serving me a cold hot dog is a suicidal strategy.
9. My name gets called. Now I know I will have my hot dog.

[FIXME: complete]

The transaction pitfall

There’s more to consistency than it’s apparent to the eye

Chapter Goals

- See consistency from a different perspective
- Designing more robust processes and keeping them user friendly
- Inject precision in our modeling process

Modeling software systems

[FIXME: finish]

18. Running a Design-Level EventStorming - 10%

Scope is different

Big Picture workshop tried hard not to focus but to embrace the whole complexity and maximize learning. Now the starting point is different: * we can assume we have a shared better understanding of the underlying domain

here the focus is on *implementing software features that are solving a specific problem*

People are different

During Big Picture workshop it is normal to have a high number of people coming from the business side (the format fits nicely also when no software development activity is planned),

Outcome is different

Knowledge or *learning* is the most valuable outcome of a Big Picture EventStorming. We also realized that the biggest constraint is time availability, so there's a known risk of running out of time before finishing, whatever *finishing* means.

But that was fine: we weren't interested in *every* hot spot, just in the most critical ones, the ones worth investing time and money do find a solution.

Here we are trying to find one or more solutions to a problem worth solving. This means that the workshop isn't over until we have something that looks like a robust candidate solution.

What do we do with the Big Picture Artifact?

- It contains a *good enough* dictionary of domain events
- It highlights - usually as a hot spot - an indication of the problem we're trying to solve.
- it shows the current level of understanding of the external contexts we're working with.

Start from scratch

This is the cleanest option, it means that we have the privilege to add more space

Work on the existing model

When does it work best?

Modeling processes, injecting new concepts on top of the existing artifact works well in small groups scenario and/or when you have the possibility of a full-day (or more than one day) workshop. In such a scenario, the memory of the big picture exploration is still vivid, so there's no need to *recall* or look back to the bug picture as a reference. Remember: the big picture was a model of *our current level of understanding*, by digging deeper into key interaction we are already making it obsolete.

Where are Events Coming from?

In a business system there are 4 main sources for domain events:

- User actions
- External systems
- Time
- other domain events (via *some form of automatic reaction*[1])

Let's examine the different situations in detail.

User Actions

A `Ticket Purchased` event might be the direct consequence of a `Buy Ticket` action, performed by a `Customer` actor.

Why can't we just use UML names?

Some readers might be disappointed with my *lack of precision* when it comes to define the key concepts of the EventStorming location. Well …this lack of precision is *intentional*: I have been working a lot with UML and case tools in my past, and I've been also *teaching* UML[2]. My problem with UML is that it doesn't help having the right discussion. In fact *using UML severely harms the possibility of this discussion to happen*; that's also the main reason why we started using ***Incremental Notation*** during workshops.

[FIXME: …]

---

[1] This is an intentionally ambiguous definition. Please forgive me, more on this later.

[2] Can you imagine the boredom? I mean, I added jokes and everything possible to make this experience enjoyable, but can you remember the time people actually paid to learn a *notation*?

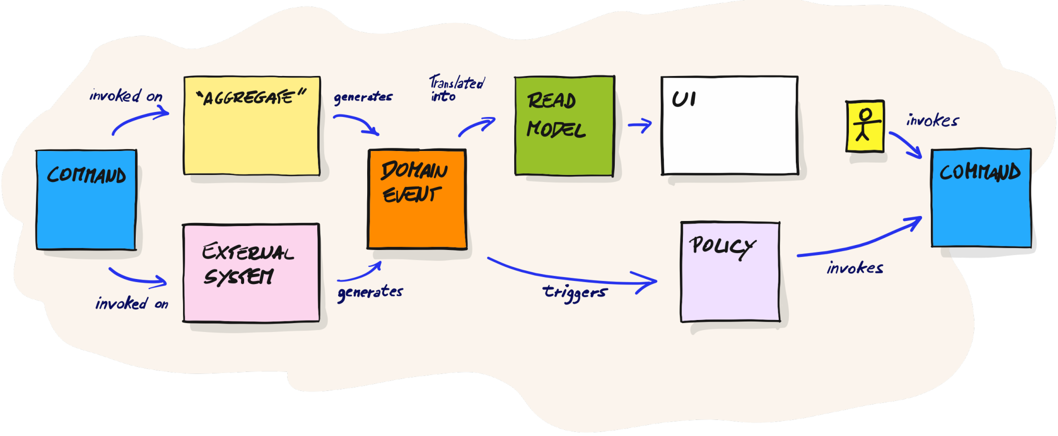
Discover Aggregates

Postpone Naming

One of the most interesting tricks is to try to postpone aggregate (or big yellow stickies, I am trying to postpone *that* too) naming. This is hard, because at this moment everybody is thinking they have a good name for it, and the *habit* of naming things is really too strong. Try the other way round: - Look for *responsibilities* first what is this yellow sticky responsible for? Which are the system's expectation towards it? - Look for the *information needed in order to fulfill this responsibility*. - Once this is sorted out, ask yourself: _"How would I call a class with this information and purpose?".

[FIXME: add image here]

How do we know we’re over?



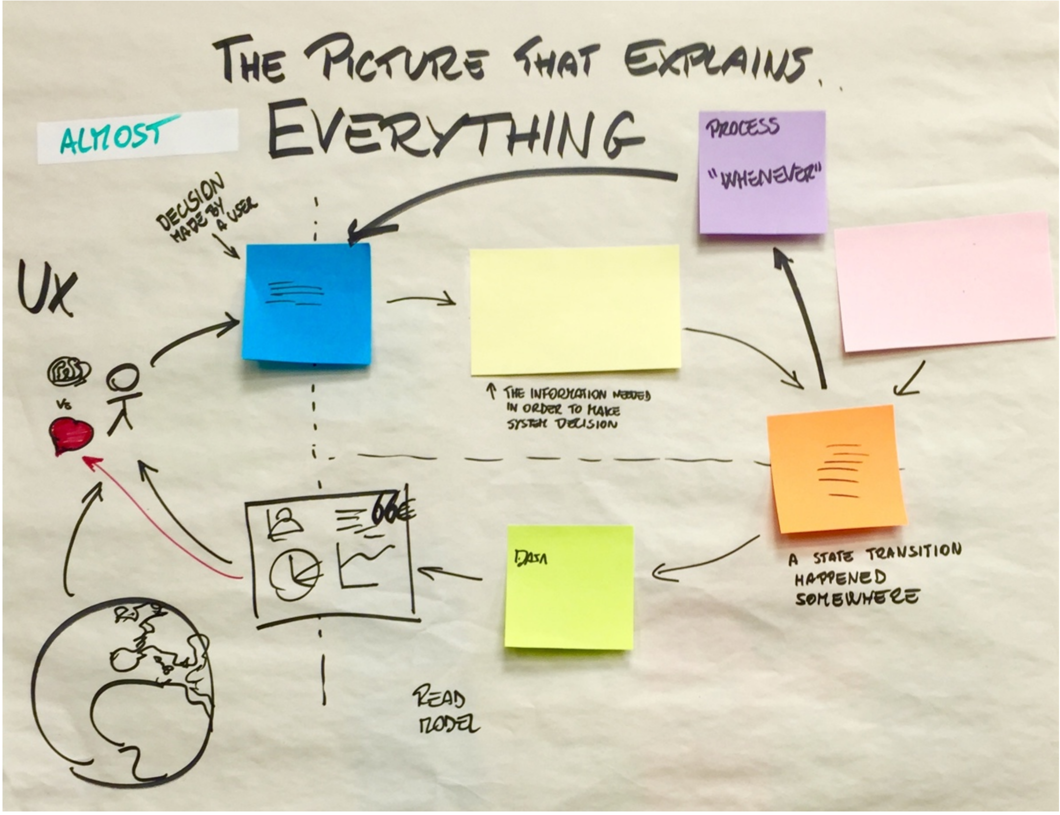
Once flattened, this is what we’re expecting in a consistent flow.

Chapter Goals

- The mechanics of a Design-Level EventStorming
- Enforcing progressive consistency
- A process not to get lost in the mess

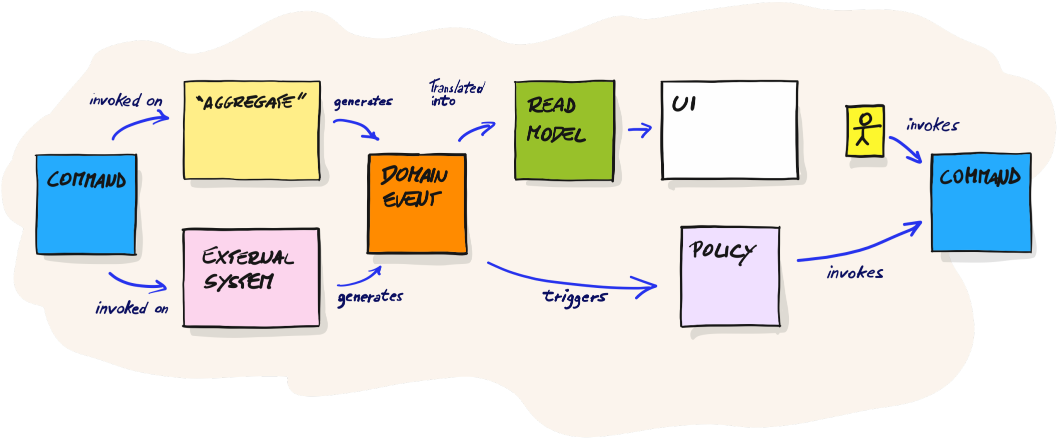
19. The picture that explains everything - 5%

a picture is worth a thousand words. This one probably a little more.


THE PICTURE THAT EXPLAINS EVERYTHING
ALMOST
PROCESS
"WHENEVER"
DECISION MADE BY A USER
UX
vs
THE INFORMATION NEEDED IN ORDER TO MAKE SYSTEM DECISION
66€
DATA
A STATE TRANSITION HAPPENED SOMEWHERE
READ MODEL

The picture that explains everything

Here we go again with the presumptuous **Picture that explains everything** in all of its splendor.

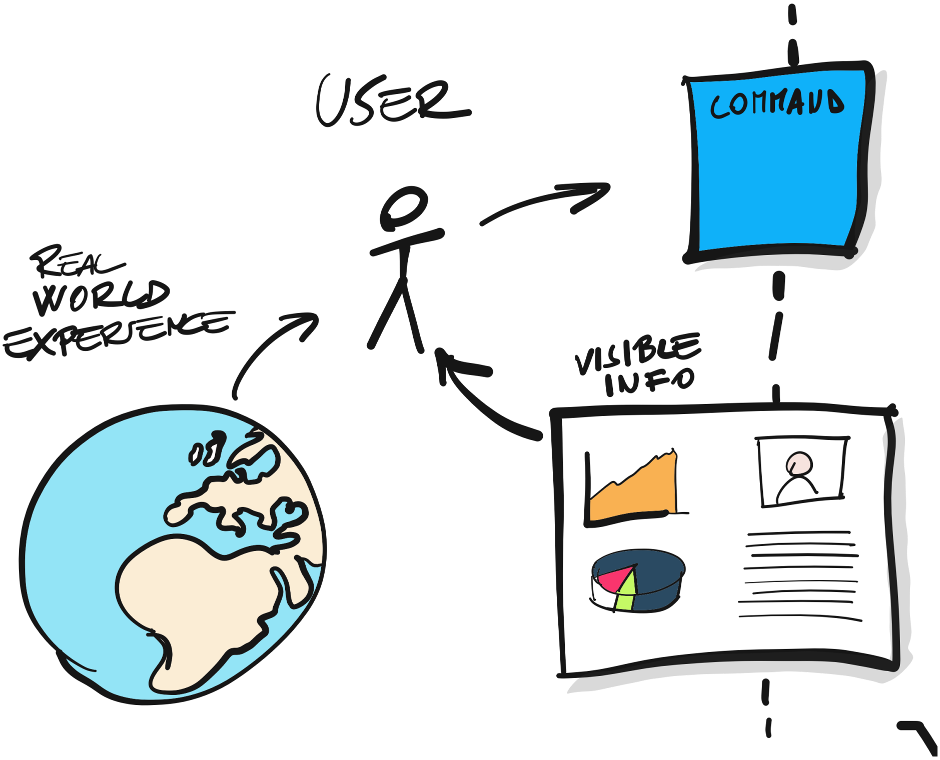


Once flattened, this is what we're expecting in a consistent flow.

20. Three types of blockers

[FIXME: maybe to be included in *The picture that explains everything*]

User Decisions



A sketch of the most common recurring patterns

Helping user decisions

Emergent Task Based UI

Nudging them

Processes and policies

What happened to aggregates?

Chapter goals:

- different type of thinking in play
- When a read model matters
- Forget CRUDs forever.

21. Design-Level modeling tips

What happens in a Design Level EventStorming is a lot different from the massive discovery typical of the Big Picture Workshop. Even if many principles are shared, here we're trying to collaboratively *design* a solution.

Dynamics are now a lot different: in GameStorming terms we're now more in a *converging* phase. We're still open to discoveries, but we'll need to agree on a possible solution.

That is going to be hard: a design session involving software architects and senior developers can easily turn into a bloodbath. Choosing a solution can quickly turn into a competition for leadership:

Make the alternatives visible

II won't promise miracles: it is not guaranteed that you'll *agree* on a solution at the end of the session. But I can guarantee that you won't agree on a solution before modeling it.

Whenever you're trapped in a Religion War [FIXME: Finish]

Choose later

Pick a Problem

After a large scale chaotic

Rewrite, then rewrite, then rewrite again.

The more you drill down into your flow, the more you'll fill the need to rephrase your events and commands.

It will look like waste, but keep in mind two things:

1. This is not software yet. The *sunken cost fallacy* will still be present in our brains, but we're only thrashing sticky notes. If we're in unlimited supply mode, this shouldn't be e problem.
2. Once released in production, Domain Events have a very annoying cost of update, due to their high number of potential listeners. Renaming a domain event in order to increase precision, might require many other software components to be updated. As every Domain-Driven Design practitioner knows very well, *naming* is an incredibly hard problem, so anticipating the mess, while the model is still only paper is probably a good idea.

So …be prepared to rewrite stickies like there's no tomorrow.

Hide unnecessary complexity

After solving a tricky problem, the solution will usually look more sophisticated than the original understanding. Our exploration led to a deeper understanding, and to discover that the underlying model was more complicated than we expected.

However, this does not mean that the *visible* model should reflect the underlying complexity. As system designers we had the privilege of a deeper exploration, but average users won't have this privilege.

Instead of exposing this complexity to the user, we might find a way to keep it simple on the outside[1]. [FIXME: finish this one]

---

[1] The obvious example is the Google search user interface: incredibly complex behind the scenes, but incredibly simple on the surface.

Events won’t be perfect but…

If you’re taking the EventStorming workshop to the extreme consequences of implementing an Event-Driven software, you probably already know that changing domain events signature, might have high ripple effect, especially if you rename things and you have multiple subscribers.

Symmetry might not be your friend after all

During the first round of coupling commands with Events

Internal and external events

Postpone aggregate naming

22. Building Blocks - 20%

Why are Domain Events so special?

Domain Events are easy

I’ve seen too many dysfunctional meetings where developers spent time trying to make sense of their own UML diagram while other participants ended up disengaging, without a possibility to provide relevant feedback on the current modeling session.

But more than this, …I haven’t seen these meetings happen anymore. Since collaboration wasn’t really happening and stakeholders time wasn’t happening in a productive way, I saw those pretend-to-be-collaborative meetings slowly disappear,

A key principle of EventStorming is to *maximize engagement of all participants*[FIXME: do I list principles clearly? Do I include this one?]. Domain Events work pretty well compared to more formal techniques with a higher threshold to join the conversation.

In fact, when kicking off an EventStorming workshop, no previous experience is required by the vast majority of participants. Yet, we can make it work really well. The secret? Domain Events are deceptively simple: *something relevant that happens in our business, written on an orange sticky note with the verb at past tense*. Even if workshop participants never had a previous exposure to modeling, they’ll quickly realize it’s simple enough to join.

In fact, I am looking for a workable definition, that becomes immediately *actionable* once the Ice Breaker writes the first sticky note). We’ll see later how domain events will help us to *inject precision* in our model, but we have to keep in mind that introducing precision too early might not be a good idea. [FIXME: more on this later]

No previous experience required

Differently from traditional modeling techniques, that referred to a specific notation (be it UML, BPMN, E-R diagrams or your favorite one) in EventStorming we are keen on *not sticking to a specific predefined notation*. The actual notation is going to be collaboratively defined on the fly, one element at a time.

This is how we keep participants engaged in the discussion: the simplest possible notation will allow everyone in the room to actively participate. In every workshop there's going to be somebody that hasn't been in a previous EventStorming: those people are probably bringing the most interesting contribution. It's our job to make sure noting gets in the way.

Every feedback is welcome!

EventStorming is an act of collaborative learning. We don't know who holds the truth. In fact we know that *nobody holds the whole truth*. The only hope

Can we make them even easier?

The term Domain Event can resonate with a specific audience. It comes originally from the Domain-Driven Design community where Greg Young and Eric Evans made the pattern originally formalized by Martin Fowler[1] popular.

---

[1] The available online definition dates 2005 on Martin Fowler's bliki: http://martinfowler.com/eaaDev/DomainEvent.html

What if Domain Events don't click with my audience?

Despite being deceptively simple, sometimes domain events don't resonate with some audiences. In a few cases, this happened because the term *Domain Event* might sound like a technical term[2], putting non technical people into a defensive mood.

It's a facilitator responsibility to enable everyone's engagement: if sticking to the term *Domain Event* turns into a problem, let's use *Fact* or *Thing that happened* instead. Agreeing on the *precise* meaning of the orange sticky may not be relevant at this stage (but odds are that there are technical people in the room for whom this is *really* important), keep in mind that in the early minutes, engagement is way more important than precision.

Anyway, providing a live example is way more useful than finding a proper word. If the audience is puzzled, the one that puts the first sticky on the wall is your best ally.

Events are precise

The importance of the verb

In my professional life, the sentence with the highest value-per-character is probably this quote from Greg Young:

"Use a verb at past tense"

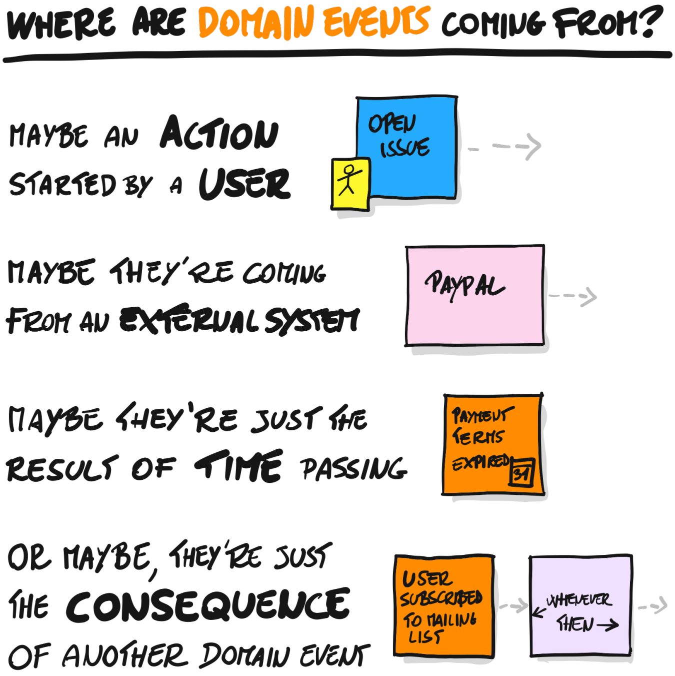
No implicit scope limitation

Events are *facts* happening in the domain. There's no implicit filter on the source: in fact, they can happen for different reasons:

---

[2] Or worse, in Italian the word *Domain* translates into *Dominio* that means also *Domination* with possibly bizarre consequences.

- they might be the *consequence* of some User Initiated Action,
- they might be coming from some *external system*,
- they might be the result of *time* passing,
- they might be the *direct consequence* of some other event.


WHERE ARE DOMAIN EVENTS COMING FROM?
MAYBE AN ACTION STARTED BY A USER
OPEN ISSUE
MAYBE THEY'RE COMING FROM AN EXTERNAL SYSTEM
PAYPAL
MAYBE THEY'RE JUST THE RESULT OF TIME PASSING
PAYMENT TERMS EXPIRED
31
OR MAYBE, THEY'RE JUST THE CONSEQUENCE OF ANOTHER DOMAIN EVENT
USER SUBSCRIBED TO MAILING LIST
WHENEVER THEN

Where are domain events coming from? Four different possible answers to the question.

In practice, this means that we're not focusing on user interaction only - like it would happen when focusing on *Commands* or *User Actions* - but ***on the whole system*** instead.

*Choosing **Domain Events** as a starting points help us to remove a blind spot*

Domain Events as state transitions

Using a verb at past tense also forces us to explore the whole domain with the focus on *state transitions*, meaning *the exact moment something changes*. This may provide a formally different semantic to our reasoning. Imagine we are collecting temperature information from an external system, a first DomainEvent candidate might be `Temperature Raised`. A closer look might show that this isn't exactly the thing that happened; in fact we might need to have a combination of `Temperature Registered` from an external source and a `Temperature increment measured` as a consequence, and realize that the initial writing, despite being correct, was actually closer to weather smalltalk than to system design.

To tell the whole story, this doesn't mean at all the initial writing is *wrong*. It's absolutely fine, especially if it triggers further reasonings. Please don't try to make it precise too early (see also Embrace Fuzziness in the patterns section).

What if my language has no past tense?

A colleague [FIXME: recover name] made me notice that the apparently trivial requirement to use a verb at past tense, could be tricky to implement in some languages. For example, Mandarin Chinese has no past tense. I can't point to a specific solution in this case, but my recommendation is to make sure that

Domain Events are triggers for consequences

The forest and the trees

A common question when explaining domain events is: *"Aren't we going too much into details?"* People are usually concerned that an event-level analysis would pull people down the rabbit hole and miss the big picture.

There's clearly a facilitator duty here in softly managing diverging conversations (more about it in [FIXME: reference]).

Domain Events are leading us towards the bottleneck

If you're into Theory of Constraints, [FIXME: finish this one.]

Alternative approaches

Start with Commands

Start with People

Focus on Key Artifacts

[FIXME: describe the document-based technique by Greg Young]

Wrapping everything up

Let's try to summarize the reasons behind the preference for Domain Events, just in case someone explicitly asks for it:

- They're *easy enough* to be grasped by everyone in the room.
- They're *precise enough* to remove a whole lot of ambiguities from the conversation.
- They're *meaningful*, by definition. If no one cares about a sticky note, maybe it's not that relevant in your system.
- They represent *state transitions* to which reactive logic can be attached.

- They can point us towards the *bottleneck* of our current flow: probably the most relevant problem to solve.
- They can *support different narratives and modeling techniques*: conversations from very different backgrounds can sparkle from the very same model.

Commands - Actions - Decisions

Blue sticky notes are the key ingredient for user interaction. They’re the result of a user decision (which might have required some sophisticated thinking) and are the trigger of some computation on the other side. If you focus on human behavior, you might see them as some action that a user is performing, like *registering on a system* or *placing an order*. If you focus on system implementation instead, the blue sticky note can represent a command you’ve received, and that your system has to fulfill.

This lack of precision in the definition of the *exact meaning* of the blue sticky note might disappoint you.

☑ **Chapter Goals:**

- pros and cons of choosing domain events as building blocks
- the potential for precision
- leverage domain events to model you progress in a lean-agile way

23. Color Modeling Patterns - 10%

Policies

Basic structure

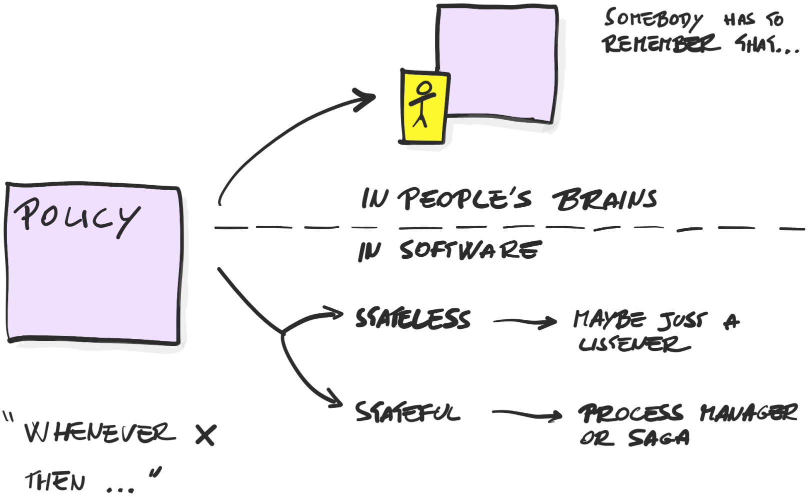
A policy is represented by a lilac sticky note, sitting in between an orange event and a blue command.

[FIXME: a three building block picture here would be perfect]

A *policy* captures the reactive logic of our processes. Verbally, we can express it like:

Whenever [event(s)] then [command(s)]

‘Whenever’ is the keyword. It helps us highlight the expected *system reaction* whenever a given event, or a combination of events, happens.



From policies to implementation

We look for policies like the missing glue between a domain event and the resulting command. Or *“there must be a lilac between an orange and the blue”*.

[FIXME: simple image]

Not every policy will be written in software. In practice, a lilac sticky captures a wide range of different implementations, spanning from consolidated habits of our organization’s employees to sophisticated implementation of distributed software processes.

The thing to look after is still the same: the repeatable reaction to a given event.

Uncodified routine

In small companies, the simplest implementation of a policy may be the usual routine of a single person. When investigating policies, we are in fact exposing their usual working approach.

The involved experts might actually enjoy having a chance to describe their own work, or might not like it at all behaving like their territory has just being invaded. That is usually the case for understanding what's going on.

Sometimes, you don't have a single person applying the policy, and different people might in fact implement the policy differently. Actually, nobody ever called it a policy. *This is just what we do!* but the moment you ask *"What happens when this event happens?"* and you get two different answers, you'll know that your investigation time was well spent.

While organizing a conference, me and other co-organizers didn't agree on a common policy for discounting: after a while, we realized that customers were in fact *polling* us and choosing the most favorable conditions.

Human Agreements

An explicit agreement is a little more sophisticated that no agreement at all.

The common verbal pattern used to describe it is *"whenever X we have to remember to do Y*[1]*"*.

'*We*' is actually the keyword, it means that we expects our colleagues to do the same.

Sometimes the agreement might be as simple as *"whenever our boss tells us to do something, we do it."* But often it will force us to discover that the implicit agreement is in fact more sophisticated: *"...I will raise a warning if the task is beyond by possibilities, or cannot be completed before the expected deadline. And of course I am not a slave, so I'll refuse tasks that are against the law or my personal ethics."*

---

[1] These little words, *'we have to remember'*, expose all the fragility of the agreement: *'we have to'* suggest the idea of a duty, while remembering is a fallacious brain task. It's not a surprise that this agreement won't scale up in non trivial organization, and won't be effective even in small ones where attention is a contended resource.

In nontrivial organizations, exploring human agreements is a continuous source of surprises: co-workers might rely on a flawed implicit agreement, that doesn't work in practice. Or you might discover that there is not a real agreement and that the system is not exposing a predictable behavior.

When inconsistencies are exposed, it's not uncommon to fix them on the fly. If the key people are in the room, and you have all the needed information, there's no reason to call a meeting: you're already having it.

However, variability in the system response to a given event might be a way to make the system more *antifragile*. Our approach while investigating policies is to doscover the hidden truth, not to normalize everything.

Being explicit, and speaking out loud will help a lot in detecting little inconsistencies.

[FIXME: something is still missing]

Below radar

One interesting side effect of looking for explicit policies

[Not every policy hides a surprise. Sometimes looking after color consistency - *there has to be a lilac between orange and blue* - will only force you to name things which are too obvious to be noticed. Something like *"whenever a user calls to cancel a reservation, I cancel the reservation"* or *"whenever we tell a customer that we'll refund him, we actually do it"*.

But the discovery of the little details is what makes the policy understood and sound. Have you ever told somebody: *"I'll do it right away."* only to pick a phone call that distracted you from the task at hand, and then rush out from the office to catch a train? Sometimes policies aren't that robust.]

In fact, when policies are too trivial, finding them a proper name might be harder than expected. No worries: if you can't find a name now, just stick with a placeholder. Often you'll find that many reactions are in fact part of the same policy, and finding an appropriate name would be obvious, but it might take a while to connect all the dots.

What about external users?

Should we model external users behavior in the same way? Some customers might in fact have a *policy* of buying ticket whenever a special discount is available.

However, while we can somehow influence our internal users, acting on external users behavior, is a totally different matter. External users will be predictable to a given extent, but it's when they're not that we might learn something useful about them. Capturing this variability with a predictable policy doesn't seem …fair.

In practice, there's not a black-or-white type of distinction. As a rule of thumb, I try to capture internal users implicit agreement with a policy, and I try to explore external users with more sophisticated personas.

But the real fun is in discovery: whenever our reasoning templates seem to fall short in capturing the underlying reality complexity, that's when a you may try a new experiment.

EventStorming is not a rule-based system. It's a platform for discovery, feel free to expand it.

Codified rules and procedure

An organization might have written rules or protocols. They *have to* follow them. Think what happens when a patient enters a hospital. In these scenarios you won't probably discover much, but developers will probably learn a lot.

Reactive logic in software

How long should we postpone technology?

During the initial phases of modeling, I often try not to use technical terms,

Visible legend

Whatever strategy you use, your best ally is to provide a visual match between the definition *“a Domain Event is something relevant happened in the system, expressed in past tense”*, and a vivid example in a visible legend. During modeling, the color would be more important than the name we’ll be using.

Language drift

In an EventStorming workshop, people with different backgrounds are collaborating to model one or more business processes. Different backgrounds often mean *different ways to name things*: software architects, user experience designers, and business stakeholders have been practicing models with different school, with very distinct flavors.

The moment people will recognize a piece of familiar notation, will try to *stick on the definition they know*. Something like: *“Okay, let’s agree we’re calling them actors from now on.”*

I just don’t do that. I aggressively try not to stick to any specific biased definition, except what is currently visible. A precise notation, such as UML or BPMN, will introduce unnecessary implicit concepts in the background: things which are going to be *clear* for some participants and *totally obscure for someone else*.

So I stick to the visible color-based definition instead which I show in the Visible Legend, which is not precise at all, but is easy enough to invite everyone to join the conversation and allows for inter-tribe communication. I *do know* that an Actor isn’t *exactly* a Persona, but during the workshop I just don’t care.

See also embrace fuzziness

Here’s a little schema that might help…

Business –> Software Design –> DDD/CQRS

Orange: Fact –> Domain Event
Blue: Decision –> Action –> Command

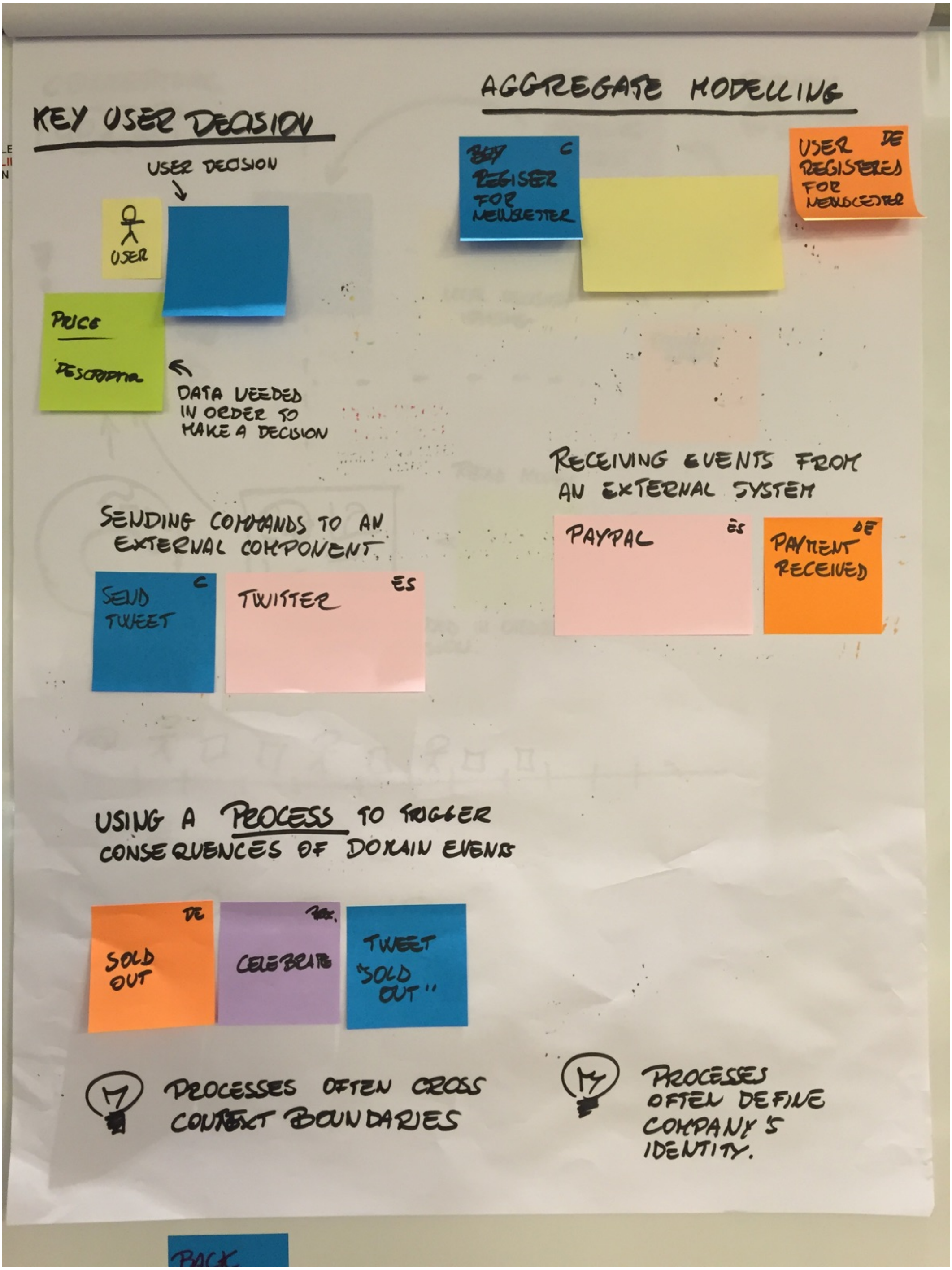
Lilac: Policy (‘Whenever’) –> Reactive Logic –> Listener, Process Manager (or Saga)
Yellow (figure): Actor/user –> Persona –> Role
Yellow (large): ? –> Local Logic –> Aggregate
Green: Key information –> Read model

[FIXME: a picture would be way better here]

Why Orange?

[FIXME: placeholder for the color picking story].

Time Triggered Events


KEY USER DECISION
USER DECISION
USER
PRICE
DESCRIPTION
DATA NEEDED IN ORDER TO MAKE A DECISION
AGGREGATE MODELLING
C
REGISER FOR NEWSLETTER
USER REGISTERED FOR NEWSLETTER
DE
RECEIVING EVENTS FROM AN EXTERNAL SYSTEM
PAYPAL
ES
PAYMENT RECEIVED
DE
SENDING COMMANDS TO AN EXTERNAL COMPONENT
SEND TWEET
C
TWITTER
ES
USING A PROCESS TO TRIGGER CONSEQUENCES OF DOMAIN EVENTS
SOLD OUT
DE
CELEBRATE
TWEET "SOLD OUT"
PROCESSES OFTEN CROSS CONTEXT BOUNDARIES
PROCESSES OFTEN DEFINE COMPANY'S IDENTITY.
BACK

Chapter goals:

- Why colors are powerful
- What can we do for the visually impaired
- A different style of reasoning

24. Modeling Aggregates

In *Domain-Driven Design*, Aggregates are defined as units of transactional consistency. They are groups of *objects* whose state can change, but that should always expose some consistency as a whole.

This consistency is achieved enforcing given *invariants*, properties that should always be true, no matter what.

Consider the simple example of a `ShoppingCart`[1] for an on-line shop: we might want to *add* or *remove items*, and obviously we expect to have the items count, and the amount subtotal to reflect the current status of the cart. What we clearly don't want, is to allow users to change the quantity of a given item, without affecting other calculated values.

The actual invariant could be expressed like:

> *The Cart subtotal will always be the sum of each article quantity multiplied by the corresponding unit price.*

[FIXME: Explore math notation in markdown]

In other words, in our software there would be no way to have inconsistent reads while accessing different portions of the aggregate. A `subtotal` property or a `getSubtotal()` method of our `ShoppingCart` will always return a value consistent with the current list of `items` in the current `ShoppingCart`.

If developers can play by these rules, they can then decide whether to calculate values on the fly, *whenever someone is accessing it* or upon change and then storing the result in some variable. Aggregates are in fact an elegant way to define a behavioral contract with a given class, providing freedom of implementation on the internals. Some may call it *encapsulation*. ;-)

---

[1] May the gods forgive me for choosing a trivial, boring, somewhat simplified and over used example.

Discovering aggregates

There is no mandatory way to discover aggregates in Domain-Driven Design, but of course I have my preferences.

Looking at the *data* to be *contained* in the aggregate isn't the best way to go. We've seen how data and the static structure of a class would drive software stakeholders into *misleading agreements*: everybody would pretend to agree we need a container for the data, and the data will need to be used (or *reused*) in many places.

Let's see this treacherous thinking process in action.

A shopping cart must be associated with a Customer.

Sounds obvious, …but it's not entirely true. In practice, a `ShoppingCart` will need to be associated with a `WebSession` that may be associated to an authenticated `User` *…or not!* We'll need a `Customer` in order to create a valid `Order` starting from the `ShoppingCart`, and this would force the current `User`s to log in if they started adding items in some anonymous session.

A shopping cart will include the list of the items to be purchased, with the associated quantity and price.

Bread and butter, apparently, except that we now should be asking ourselves whether we need to include the `ItemDescription` in the `ItemInCart`. Feels like we should, because we'll need to display the `ShoppingCart` info to the customer, in order to review the cart before proceeding to checkout *"is this really the stuff you intended to buy? Have you looked at the grand total?"*. Things might get awkward when starting to consider events like `ItemPriceUpdated` or `ItemDescriptionUpdated`, that should have us thinking whether we should include a copy of the entire description of the selected item, or just a reference to the `Item` in stock.

Hmmm… not. Sorry for the detour, but these are not the aggregates we're looking for. *"data to be displayed to a user in order to make a decision"* will be a **Read Model**. Aggregates are something else, but we have to be aware of this vicious temptation of superimposing *what we need to see on the screen* on the *internal structure of our model*.

They're not the same thing.

Aggregates as state machines

What I am really looking for are *units of consistent behavior*, and given all the corporate dysfunctions that we mentioned in chapter 2, I am not expecting consistency to scale much.

25. Event Design Patterns - 5%

[FIXME: Or should it be Event Discovery Patterns?] [FIXME: candidate to be killed]

Discovery strategies

Most significant events have what tech guys call a *“high fan-out”* meaning that the number of *listeners* or *consumers* of a particular Domain Event is higher than the number of producers.

In our *public training* domain, a `TicketSold` event might be interesting for many departments: classroom capacity will need to be updated, as well as numbers for lunch and coffee break; an eventual loyalty management system will be interested and the trainer will probably like to be notified as well.

During EventStorming exploration, I prefer listening to the needs of the downstream domains:

Is it a domain event?

The term *Event* is overloaded: there are

Composite Domain Event

Sometimes

Chapter Goals

- Domain-Driven Design wording obsession in practice
- Sometimes it is necessary to be redundant
- Overcoming some common stalemates in Event-Based modeling
- Closing the loop with Enterprise Integration Patterns book

26. Collaborative modeling tips and tricks

Modeling a complex piece of software collectively, poses a series of challenges that we might not be prepared to.

Notation and colors provide a blueprint about how a plausible model might look like, but provide little guidance about how to get there. However, the biggest challenge is probably that collaborative modeling exposes your thinking process and the intermediate steps to your peers, who might have a completely different approach.

Here are a few tips that might come in handy during a Process Modeling or a Software Design session.

Model the right problem

Keep the model clean

Postpone branches

While exploring a process or a feature, suddenly things start feeling harder than necessary. Many times, you might find yourself discussing a colleague's proposal in terms of *but it won't work if [put your scenario here]*.

Unfortunately you won't get very far trying to solve everything at once. Every alternative is making the problem harder leaving you juggling with a continuously growing solutions space.

A good idea might be to make sure we're always only solving a single problem.

Finish First / Correct Later

Modeling out loud

Your verbal brain is your best ally during a design session. But your brain is not as trustworthy as you might think: it's basically *lazy* and will try to solve problems with minimum effort, whenever possible[1].

The best way to force deeper reasoning, is by engaging multiple areas of the brain, turning the discussion into an *announcement*: *"we have a* `PaymentReceived` *here*, *and our* `Billing Policy` *is to generate a receipt whenever a payment is received. Then whenever a receipt is generated…* " We need a policy here.

Brainstorm on terms

[FIXME: image of a term pool]

Brainstorm on context

Before feeling too proud for your emerging model, you might want to check its robustness, by adding a little spice on your policies: adding words like *always* and *immediately* to the policy implementation usually triggers some afterthought, especially if the policy is implemented by humans.

Summary

[FIXME: Here a visible map of all the forces in play might come in handy.]

---

[1] This is probably the shortest possible summary of Daniel Kahneman's book *"Thinking fast and slow"*.

27. From paper roll to working code

Managing the design level EventStorming artifact

After the discussion with the small group when you decomposed the problem in Commands, Events, Aggregates, Processes and Read Models, you have an artifact which can be used as a living support for implementation. The visible model provides some interesting information in form of visual pattern matching, colors provide some form of balance and an easy symmetry. It's not infrequent for domain experts without knowledge of the notation to jump in and ask: "Why is this orange thing without the yellow one?" The threshold for contributing to the discussion is way lower than with traditional UML.

Modeling interactions with Event-Driven CRC cards

The great advantage of EventStorming is that it promotes a collaborative form of modeling, while the whiteboard may be inclined to a "let me show you" type of approach. However, when design starts to focus around roles, responsibilities and message passing, a very efficient way to model interaction is to simulate behavior with a variation of the CRC cards. - Humans may take the role of Users, Aggregates, Processes and Projections. Put in another way: humans are representing the decision makers in the system. - Commands, Domain Events and UIs are represented by cards. Cards will carry the information written on them. To model a process, every human can produce an output only based on the information available. To make things more interesting, and to model the system in a tell don't ask fashion, humans can't ask, only tell. This way we can normally sketch the necessary communication patterns needed for a component to work seamlessly in an event-based solution.

Coding ASAP

As colorful as it could be, a post-it based model *does not compile*, nor it delivers a green bar. We shouldn’t put too much emphasis on doing EventStorming right:

the roll is not the deliverable, it’s just a way to get to the right implementation faster.

So as soon as you have a reasonably good idea about the underlying model… start coding it. There’s nothing better than a good prototype to discover flaws in our reasoning. Depending on your existing infrastructure, writing a simple prototype can also be very fast, at least for the domain model part.

Being on a fast-track

We must not forget that the whole cycle started with the assumptions of discovering what was the most urgent matter for the whole organization. If that problem called for a software solution, there are very few reasons to slow down.

Even if the workshops are conceptually divided in stages,

28. From EventStorming to UserStories - 5%

A placeholder and a better conversation

Defining the acceptance criteria

Events

Read Models

User Interface

This is where things gets trickier: both Domain Events and Read models had a *black-or-white* completion criteria. With user interface, this is not always the case. Users need to find the UI *usable* or *beautiful*, [FIXME: finish this one]

It’s no secret that the original article [Link] from Jeff Patton had a great influence in the way my brain works. To be honest, more than the article, it was the pictures. They were using a huge amount of space. They were using all the space needed.

EventStorming and User Story Mapping

It’s no secret that the original article [Link] from Jeff Patton had a great influence in the way my brain works. To be honest, more than the article, it was the pictures. They were using a huge amount of space. They were using all the space needed.

However, having had a chance to peek into

Scope

The first main difference is in scope. User Story Mapping focuses in the understanding needed *to develop a new product*, while EventStorming has a broader scope, not necessarily tied to product development.

Tasks first

User Story Mapping starts from User Actions

Tasks / Actions are easier to grasp

Domain Events have a broader scope

Focus on the Minimum Viable Product

How to combine the two approaches?

EventStorming and User Story Mapping both leverage the availability of key experts and decision maker in order to trigger meaningful conversation. *A lot of there conversations will be the same* and, to be brutally honest, this is more due to the dysfunction of the existing approaches than to the merits of the two formats.

However, some parts of this conversation will

[FIXME this calls for a Venn like diagram.]

Chapter Goals:

- it’s all about the conversation

- being sequential is actually suboptimal
- too many good ideas in the same space

29. Modeling in the large - 10%

Nobody taught us how to design large systems, we just designed small ones and then bloated them.

Enter Bounded Contexts

The idea of Bounded Contexts comes straight from Domain-Driven Design and it's one of the most powerful and neglected concepts in software development.

Multiple collaborating models

Bounded Contexts modeling patterns

After seeing many companies and business processes in different domains, I realized there were some interesting similarities in the way different domains can be partitioned in bounded contexts, and also in the way similar bounded contexts can be modeled.

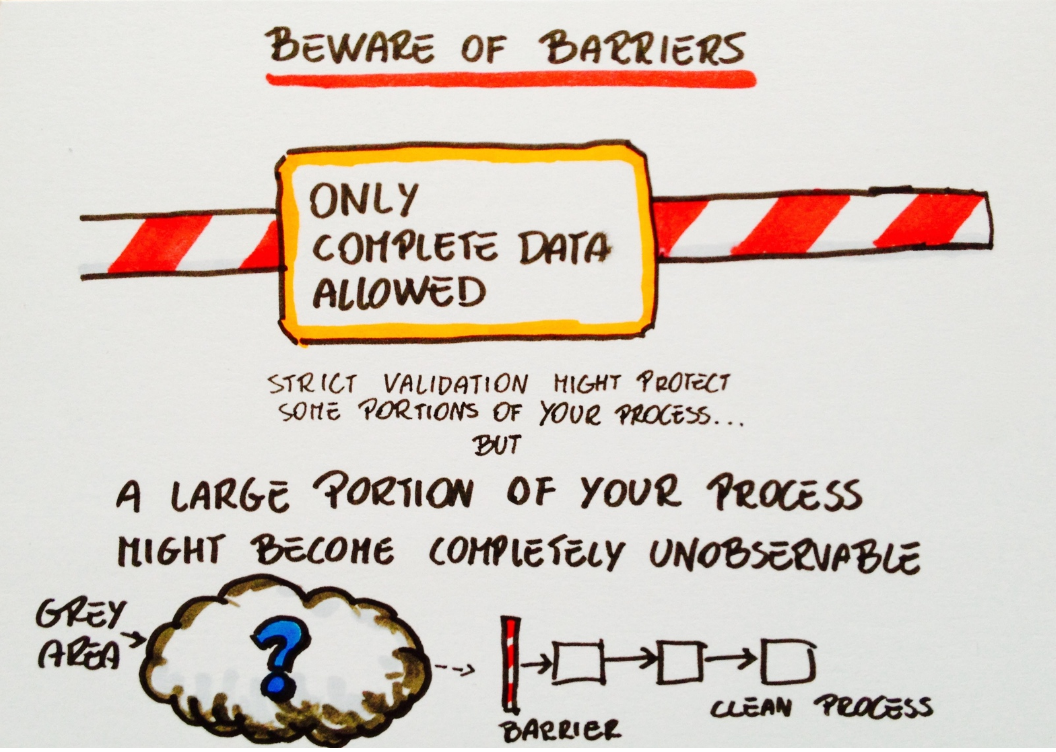
I also realized that there's not so much literature about it. The only book that comes to my mind is the old Martin Fowler's *Analysis Patterns*, that unfortunately predates the idea of bounded contexts.

Hypocrite modeling

A common mistake that happens in modeling business processes happens when we're trying to pretend that reality should be subject to our rules and not the other way round.

Suppose we're writing management software for our little company. In order to be billed every Customer should have a VAT number (at least, according to Italian regulations). Validation zealots will then add the `vat_number` field to a form and make it mandatory.

However, in real life, we have plenty of situations where we might want to store a 'Customer' somewhere even if we don't have the VAT number yet (working with startups is great fun when it comes to billing). If the VAT Number is mandatory for our management software, here we are! Welcome to the hell of information spreading!


BEWARE OF BARRIERS
ONLY COMPLETE DATA ALLOWED
STRICT VALIDATION MIGHT PROTECT SOME PORTIONS OF YOUR PROCESS...
BUT
A LARGE PORTION OF YOUR PROCESS MIGHT BECOME COMPLETELY UNOBSERVABLE
GREY AREA
?
BARRIER
CLEAN PROCESS

Hypocrite modeling normally shows up as validation

Collaborative Construction

Chapter Goals

- There are recurring patterns for modeling in the large
- Sharing a database is evil
- Data-first modeling will lead you to the wrong design

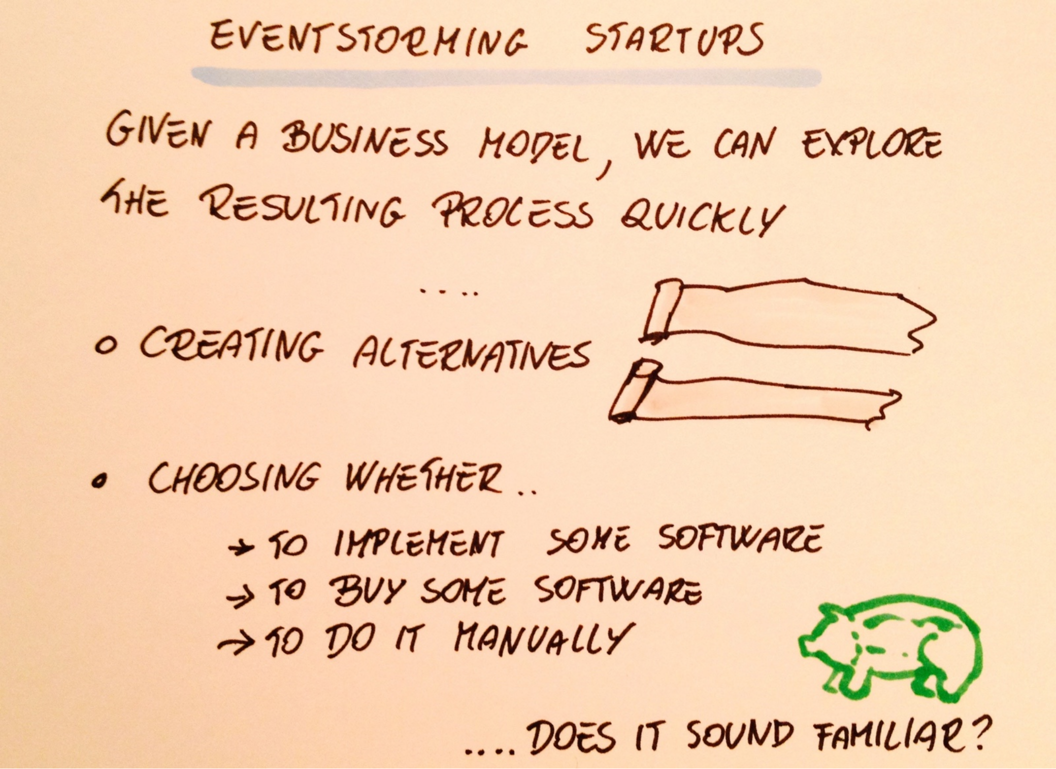
30. Working with Startups - 2%

- too early to have corporate silos;
- people are more likely to admit they don't know something;
- there are things that nobody in the room knows;
- not so many competing activities: creating a cool product-solution is the most important thing to do.

The focus is not on the app

Leverage Wisdom of the crowd

Multiple business models



EVENTSTORMING STARTUPS

GIVEN A BUSINESS MODEL, WE CAN EXPLORE THE RESULTING PROCESS QUICKLY

....

- CREATING ALTERNATIVES
- CHOOSING WHETHER..
  - → TO IMPLEMENT SOME SOFTWARE
  - → TO BUY SOME SOFTWARE
  - → TO DO IT MANUALLY

....DOES IT SOUND FAMILIAR?

Explore multiple business models, the results might suprise you.

Chapter Goals

The coolest app isn't enough EventStorming is second best Validate the assumptions Learn, and the learn more

31. Working in corporate environment - 5%

Invitation is crucial

The privileges of being a consultant

Being *'the EventStorming guy'* provides certain advantages over the normal developer within a corporation. Many times I am invited in a company *just to run an EventStorming session*, so that I have some degrees of freedom in asking special requests for the workshop day, and I leverage it. After all, I'll be there only for a given amount of time, and wasting that special day seems really stupid.

But that's my privileged position of being a consultant coming once in a while. I try asking for 'all the key stakeholders', and I may get them. But I know that if I was a normal employee instead, things would not be that easy.

So, keep that in mind, and be prepared to a softer, slower, iterative approach when managing invitations as a corporate employee.

Convincing is waste

Be aware that *running an EventStorming workshop* is nobody's goal. *Discovering key impediments* might be somebody's goal. Even *learning about the domain complexity* isn't necessarily a selling point, since the experts you are try to convince might not consider themselves in need of learning: *they're experts!* Learning is your problem, not theirs.

However, if you're spending a lot of time describing what is going to happen in a workshop, keep in mind that if will be really hard to make a compelling case just talking about it. Just doing it is by far the better strategy.

Of course, if your experts have spent countless hours and long boring meetings in order to produce a specification document you might have just one more reason not to run the workshop: *"I've already spent a lot of time writing the specs, can't you just read them?"*

[FIXME: Finish]

A trap fro domain experts

The best way to hunt for domain experts is to set up a trap for them. EventStorming is actually close to perfection in this sense, a *big colourful artifact placed in a visible location* will attract domain experts like honey attracts bears.

To make the trap [FIXME: Finish]

Invitation as an iterative process

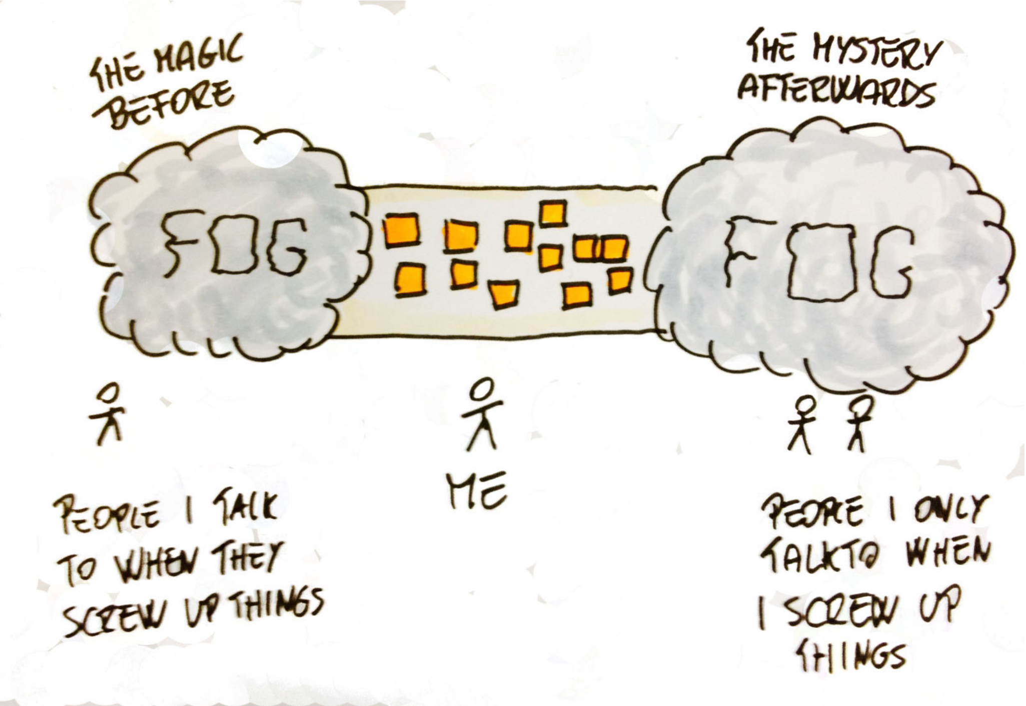
The harsh reality is that people will understand what EventStorming is only after practicing it. This creates a stalemate with no easy solution, unless we break the assumption that we have to run only one session.

As a consultant I might try to optimize costs by running avery intense and brain demanding

Mixing high and low intensity

Manage check-in process

The fog-me-fog model


THE MAGIC BEFORE
THE MYSTERY AFTERWARDS
FOG
FOG
PEOPLE I TALK TO WHEN THEY SCREW UP THINGS
ME
PEOPLE I ONLY TALK TO WHEN I SCREW UP THINGS

The fog-me-fog model

Working in corporate environment - 5%

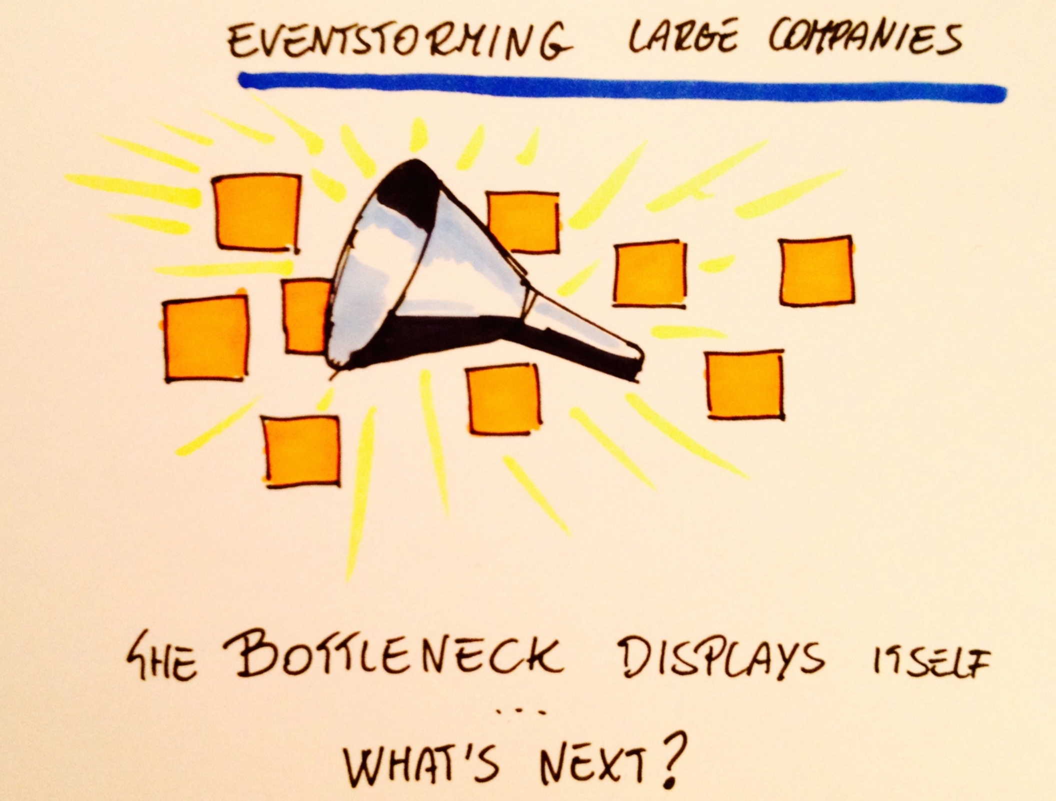
Nobody wants to look stupid

Wrapping up

What happens next?

Corporate Dysfuctions

Managing the bottleneck



The bottleneck, revealed

Deliverable Obsession

Can’t we have it remotely?

Chapter Goals

- Calibrate expectations
- Politics is king
- We’re still humans, after all

32. Designing a product

This is not a tailored solution

The bottleneck might be somewhere else

Matching expectations

Simplicity on the outside

Adjustable complexity

Flexible policies

33. Model Storming - 0%

☑ Chapter Goals:

- Understanding the difference between Model Storming and Event Storming
- Having a glimpse of what can be achieved when the rules are released
- Seed the will to experiment

34. Friends and ncestors

[FIXME: this will maybe fit in a separate chapter]

Ancestor and relatives

The idea of bringing the key people together isn't new or particularly original. The first source of inspiration I can recall is the **Joint Application Design Workshop**[1] one of the initial steps of DSDM[^DDPS3]. I remember Craig Larman suggesting it as an approach to kick-off scrum projects, but I guess I was even more fascinated by the pictures of the real workshop with plenty of people and plenty of space available and a distinct feeling of *energy*.

A big inspiration comes from David Sibbett's book Visual Meetings

Context Mapping

A hidden gem in Eric Evans' book *Domain-Driven Design, tackling complexity at the heart of software* is a tool called Context Mapping.

Impact Mapping

More recently, two approaches emerged, addressing a similar problem space.

Gojko Adzic's **Impact Mapping** aims to rediscover requirements for software to be developed by looking at the real reasons and goals behind it: there is no 'implicit acceptance' of software requirements but a process of *"tell me what the real goal is and we'll find the best available trajectory to get there"*, with the notable outcome of turning *executors* into *problem solvers* and to transform software development into just one of the available options.

---

[1] https://en.wikipedia.org/wiki/Joint_application_design

User Story Mapping

Jeff Patton launched the idea of User Story Mapping a few years ago (the blog post *The new User Story Backlog is a map*[2] dates 2008). I loved the article, but I think I was hit by the subliminal message of this picture:

This was the image that hit me. The guy was using all the space he needed.

User Story Mapping has a few similarities with EventStorming:

- it puts *learning* at the center of operations;
- it *uses all the space needed* in order to understand the problem;
- it *triggers the right conversation* involving all the key stakeholders in a software project.

While reading Jeff's book I had this feeling of *I want to steal these two chapters and put them in my book*. The mindset behind is largely the same.

However, we'll see that the goal doesn't overlap exactly. There's something that is unique to each tool, and in fact they complement each other quite well.

Value Stream Mapping

I had the chance to discuss the idea with Stephen Parry, and he's actually doing something similar: bringing key people together in order to visualize the existing system. This is part of his Lean approach, and visualization plays a big role in lean in general.

I also took some ideas from Mary and Tom Poppendieck's book Lean Software Development about visualizing Value Streams in an agile, lo-fi fashion.

[FIXME: Image of the red-green VSM]

---

[2] http://jpattonassociates.com/the-new-backlog/

However, when I got deeper into the Value Stream Mapping topic, I felt like I've bumped into another disappointing precise notation.

Similar ideas which I don't know

We we're talking about EventStorming in a London pub, and a colleague told me something like: *"we've been doing something like that in the eighties, while designing a very large system."*

Overcoming space limitations in order to design large systems isn't a very original idea, and definitely isn't *mine*. But if you have similar experiences, or if you tried already something like that in the past, I am eagerly curious to know about it.

What makes EventStorming distinctively different

When comparing EventStorming to other large scale workshops involving key people, I think the really distinctive factors boil down to:

- **Domain Events as building blocks**: apparently we're just having a conversation about a big system, but *domain events are making that conversation more intriguing*. There's a whole chapter dedicated to that: Why are domain events so special?.
- **Incremental Notation** allows the exploration to be open ended, there's not a mandatory goal to pursue, the outcome could be better than the plan if we discover something more interesting for the mix of people in the room.
- **Fuzzy Notation** is a subtle enabler for cross-discipline conversation. Choosing not to go for an explicit notation, we're not excluding anybody from the conversation. More on it in Embrace Fuzziness.

This combination makes EventStorming less a tool for gathering requirements for writing software in a complex domain, and more *a platform for cross-discipline exploration* of a complex domain.

35. Remote Event Storming

This section intentionally left blank

Ok, seriously

Honestly, I didn't want to write this chapter. I've been asked about "remote EventStorming" many times, and my answer was invariably

There is no such thing as Remote EventStorming

… and no. I haven't changed my mind since then.

But I've realized that this is a legitimate beginner question, and providing a

User Experience

I've been doing Google Hangouts, and conference calls, and Skype conversations. I still do a lot of them. Sometimes my microphone is not working. Sometimes is somebody else's headphones. Sometimes we don't know exactly which one is not working. Sometimes we switch back and forth different tools for minutes just to get the conversation started.

Yes, I am sure you have more professional corporate tools to manage online meetings. But let me tell you one thing: *every single tool introduces some friction*. And in designing the workshop we put so much attention in removing every friction point (remember *one man/one marker*?) in order to make the relevant conversations possible, that switching to an online tool feels like toothbrushing before eating marshmallows.

No Big Picture

Tool vendor will try to impress with cool features like zoom and an infinite board size. But there's one thing that you have to keep in mind.

Your screen is small

Which boils down to a sad conclusion:

You can't have Big Picture and Readability at the same time

No People interaction

Wrong attitude

No parallel conversation

Screen size is not the only

Downgrading expectations

Probably, what’s disappointing for me is that people keep talking about “Remote EventStorming” which sounds like *blasphemy* to me. If the question was something like *“do you know any cool way for remote modeling?”* then the answer may be “of course I do, “

Patterns and Anti-patterns

“The 7 most expensive words in business are: ‘We have always done it that way!’“

Catherine DeVrye

36. Patterns and Anti-Patterns - 75%

Add more space

(Pattern)

Even if we start the workshop with the *promise* of unlimited modeling space, we will quickly hit the boundaries of our modeling surface.

Then, something unexpected will happen: everything will start to feel harder and people will loose momentum and speed. This happens because your brain is actually trying to solve too many problems at the same time: *modeling the next piece of the system* and *finding an empty spot on the modeling surface*, or *moving other portions of the systems without making a mess*.

[FIXME: Picture here]

Unfortunately, our brain is not very good in framing this situation: you'll probably feel the stress of getting stuck in a harder problem. In fact, the problem isn't harder than the ones you just solved. It is the extra burden of *modeling the problem* **and** *finding an empty spot on the surface* **and** *moving the surrounding portions of the model* **possibly** *without breaking the timeline* that is making your brain's work harder.

therefore

Providing more modeling surface will break the stalemate. Usually this means adding another paper strip below the original one, or rehashing existing stickies in a more readable way.

Suddenly, modeling the big mess becomes an easy task again, since we removed the extra constraint of modeling the whole process *within the*

available space, while your brain was silently screaming: *"The solution can't fit in there!"*
In general, it's a good idea to have some extra space to add on demand. Using all the available space from the beginning - like it may happen when you have a huge fully writable wall - might create a weird feeling later, and a stalemate that can't easily be overcome.

Remember we're exploring: it's hard to see in advance how big the problem will be.

Here are some recurring scenarios where **add more space** will prove worthwhile:

- Doubling the modeling space in order to see both *current* and *desired* state of the system (and highlighting the differences)
- Keeping the *Big Picture* modeling artifact, and starting a *Design-Level* on a different scale.
- Exploring alternative paths for a business flow.
- Choosing between two different solutions, in a process modeling, or a software modeling scenario.
- Splitting the modeling team in two, when it's hard to fit different modeling styles on the same surface.

Be the worst[1]

(Pattern)

In every team, in every meeting, nobody wants to look like the most stupid person in the room. Sometimes, being that person can torn into a good facilitator's trick.

Especially when it comes to place the first domain event on the modeling surface, an initial standoff may be very embarrassing. If no allies are helping you, just be the one to start …with a twist: instead of doing your best, with the unintended consequence of setting the bar too high, just place something *evidently wrong* on the surface, so that it's going to be easy for somebody to something better. Then throw your sticky away, or have somebody rewrite/rephrase it.

---

[1] I've stolen the name from Rinat Abdullin and his blog.

This way, you're showing that you're not aiming for perfection and that being wrong is an option, with no dramatic consequences. This trick is not a risk-free one: if the initial situation is really awkward (maybe because of the way invitations were managed, of conflicting purposes upfront), this may just add awkwardness. So, handle with care.

Conquer First, Divide Later

(Pattern)

Space constraints and traditional mindset usually force people to model making an upfront selection. The boundaries of a physical modeling surface such as a whiteboard or a flipchart are in fact turning our original goal:

we need to model this problem in order to find a solution

into

we need to model this problem ***within the available space*** *in order to find a solution*

Your brain won't explicitly warn you about this change of direction. Sometimes you'll find yourself in a problem harder that it should be (see also Add more space about it). Some other times you'll quickly cut off some portions of the problem that don't look relevant.

Even more annoyingly, if the problem is *too big to fit in the available space* people just don't model it: they just *talk* about it. And this creates a vicious loop of endless conversations about what the nature of the problem is. If you find yourself trapped in a conversation about *"what the real problem is"* you know exactly what I mean.

[FIXME: image with balloons]

[FIXME: finish this one]

Do First, Explain Later

(Pattern)

When performing new activities, people usually feel the need for detailed instruction, in order to do things properly. The need is real, but it doesn't mean that satisfying the need is the best thing to do.

If your goal is swimming, understanding the theory in advance won't help much. In fact, it could make things worse: more concept with no connection yet with the actual feelings - can you explain being underwater to a person that's never been underwater? - will pile up in your brain waiting for an explanation. The more concepts piled up, the more messier would be their understanding.

therefore

Have people get in touch with the problem first. Provide just the basic information needed to have them try, and provide clarification only as a response to specific questions.

Do not create a big theory in advance: keep the time to explain the whys later, after people had the time and the feedback to create a more solid mental model[2].

It is funny to see how *the very same words* will slip away from the listener brain if told *before* the experience, and will stick instead once the experience formed an incomplete model with compelling questions.

---

[2] Reading *Pragmatic Thinking and Learning* from Andy Hunt was a big eye opener for me. A more thorough approach on the matter can be found in *Training from the Back of the room* by Sharon L. Bowman.

Fuzzy Definitions

(Pattern)

Software developers are often obsessed with terms *precision*. This is remarkable, because *ambiguity does not compile, and doesn't pass tests either*.

However, there are situations where this obsession for precision might just get in the way of a successful conversation:

- in a Big Picture EventStorming, the different perspectives *have to* clash. Enforcing precision too early in the exploration phase might exclude interesting dissonant voices from the conversation.
- Adherence to a specific conceptual model upfront will exclude what's not part of that model, which can be relevant.
- The distinction between a *User* and an *Actor* or a *Persona* isn't really that interesting, there are more interesting things to discuss, during an EventStorming.
- Choosing a specific model/notation will also favor one party over another one.

therefore

Clearly state that you won't provide precise definitions *on purpose*, ant the goal is to include different viewpoints in the same conversation.

The apparently agreeable sentence *"Ok, but we have to define what an actor is"* could be addressed like: *"Sure, we will!"* But now there are more interesting things to do. However, if the pressure for a definition is driven by clashing interpretations, then make both interpretation visible: maybe you're just discovering that you have two actors behaving differently in that given portion of the flow.

During the workshop, I'll let a visible legend be the explicit reference for the current meaning of a piece of notation. At the same time I'll try to dumb down the language up to the *"we need a blue one after a lilac one"* instead of the more precise _"we always need a command as a consequence of a policy".

Enforce the Timeline

(Pattern)

Guess First

(Pattern)

In many conversations, the learners quietly listens to the experts explanation, like in a traditional lecture.

unfortunately

This interaction has many drawbacks:

- it may be really *boring*;
- it may be *hard to interrupt your boss* during an explanation, especially if you're afraid to ask a stupid question;
- an expert usually tends to skip obvious topics, *which might not be obvious* to other participants;
- misunderstandings are not visible: the current displayed model is the official one, with little or no contradictions.

In general, better learning happens when we're conquering our knowledge with experience, not with an explanation.

therefore

Encourage the newbies to openly *guess* what the system is supposed to be. Learners can contribute to the model adding things that might be right (some developers are really good in *figuring out things that aren't written in the official specifications*), or wrong.

When the guessing is wrong, usually somebody will correct the statement. In this case the little lecture will be needed and *appreciated*: discarding the initial guessing, we create the curiosity space for the expert explanation.

hot spot

(Pattern)

Understanding how a process works is not a linear process. The larger the topic, the higher the probability you'll run into bumpy situations like:

- people being *not completely sure* about what they're saying;
- different people having *different views on the same topic*;
- people warning about how messy or risky a given step is;
- endless arguments between a couple of participants cutting off the rest of the discussion.

therefore

Use Purple stickies to mark **hot spots**. They're your annotations, or better they're the shared annotations from participants to the workshop. The resulting model won't be *right* or *finished*; hot spots will contain the meta-information which is relevant to the learners.

[FIXME: images]

Icebreaker (the)

(Pattern)

That awkward moment everybody is looking at an empty paper roll on the wall, so big, so white, so empty …and nobody knows how to start! It's like being at a party when nobody is dancing! Well, if that's the first time you're running a workshop, then *people legitimately have no clue about what to do*. The best way to exit the stalemate is to have an *Icebreaker* make the first move.

After people see what they're expected to do in practice, and after seeing they can actually improve the existing moment, things will start flowing a lot more smoothly.

The best icebreaker is normally an ally among the participants. If the moment is *really* awkward and nobody is making a move, then it's probably the facilitator's duty to make the first move. However, the facilitator is a special role: they know about the format more than others, and people might start relying on the facilitator too much in order to do the right thing.

Incremental Notation

(Pattern)

A well established standard notation sets a gap between the ones that know it, and the ones that don't. It's better to keep the shared notation *as small as possible* and increment it as we go, in order to make communication fair and efficient and to promote a collaborative discovery of the model complexity. Once we hit a *plateau of discovery* (a fancy way to say that the pace of discovery slowed down) we might then add new elements to the current notation and add them to the legend.

I normally ask participants questions like *"What is the aspect you'd like to see in the model?"* or *"What's the next thing you'd like to visualize?"* in order to go in the direction that will maximize the engagement and keep the workshop flow *in 'the zone'*.

When taken to the extreme, the incremental notation approach will open the doors of Model Storming, the radical approach to model big stuff, when you have no idea what you're doing. See also Visible Legend.

Go personal

(Pattern)

Sometimes only a few participants are actively engaged. It may happen for several reasons: maybe the room setup - people managed to grab a seat before the discussion got into flow - or maybe some implicit hierarchy, preventing the people to speak out (or simply to contribute writing on stickies).

After double checking that there are no impediments to engagements (are stickies and markers enough?) you may want to pull some engagement by asking participants: *“What do* ***you*** *do?”* and to add themselves to the picture as little named actors.

Yep, that’s our little trick: some may not feel engaged by *‘the system’* but they’ll probably like to talk about themselves. However, this trick worked well a few times, so it can be used to escape awkward situations.

Keep your mouth shut

(Pattern)

Sometimes you’ll see a better option.

Sometimes the solution is *so obvious!*.

But sometimes it’s better to resist the temptation, and let somebody else have the same idea and follow. If *nobody* had the idea… well you can try with *“It might be a very stupid thing to ask, but I was wondering… “*. Remember: nobody likes to look stupid in public.

Leave Stuff Around

(Pattern)

Once the workshop is finished, you might be tempted to wrap it [FIXME: finish]

Manage Energy

(Pattern

Make some noise!

[FIXME: Finish]

(Pattern)

Mark hot spots

[FIXME: duplicated - merge] C> *(Pattern)*

EventStorming is an act of learning and discovery. But you'll always going to meet some workshop participant thinking that the goal is a *formalization* or, even worse, a *specification*.

This is not the case. The visible outcome of EventStorming is a *representation of our current level of understanding*, with no guarantee that we'll be able to understand everything within the time frame of a single workshop.

Moreover, conflicts may arise during the workshop. In fact the workshop is designed in such a way to make the underlying conflicts *observable*.

therefore

Mark hot spots with purple sticky notes. Purple is the closest you can get in the stickies world to signal *warning* or *danger*[3], and you can use it just like annotation in your notes.

---

[3] For some reason, red is not contemplated among valid sticky notes colors. This is still a mystery to me.

They'll signal that there's a problem in a given area, or a lack of understanding (you can use hot spot to mark Socrates approved known unknowns), or whatever looks necessary to signal to your future self.

The model is you team sketchnotes, and polished ones just don't work so well.

Money on the table

(Pattern)

Developers tend to *forget* the money. Mostly because it's so *boring*. However, when discovery business processes, well …we have to include money in the conversation. The problem is that when we start talking about money it tends to displace other (valuable) perspectives. Our recommendation is to do the first round in the most natural way and once we're finished do another round focusing on money.

One Man One Marker

(Pattern)

In a normal meeting room you might find just a few, probably depleted, markers. But when only a few people are writing the collaboration becomes dysfunctional: people will form circles, agreeing on what to dictate to the one that instantly became the group scribe.

This can turn a promising workshop into a really boring and frustrating session: some voices will be heard more than others, and the quality of the outcome will be disappointing.

therefore

To ensure modeling will happen at the maximum possible speed, you have to remove impediments to parallelism. Provide enough *working* markers for everybody, so that access to key resources is not a problem.

It's a good idea to double check your markers before the workshop (and please throw away the nearly depleted ones) and to bring an extra supply, just in case. [FIXME: one man one marker image]

Poisonous Seats

(Pattern) [FIXME: it's apparently the contrary]

There are a few problems with seats. 1. Despite the appearances, sitting is a *hardly reversible action*, it takes little energy to seat, and a lot more to stand up. 2. Once sat we'll have a lot less interaction with people far away from us: interaction becomes too much influenced by our position. 3. Sitting is a lot more prone to distractions such as switching on the laptop, checking e-mail, etc. 4. Blood circulation is affected too.

therefore

Remove seats for short workshops. People joining a room with no seats.

For longer workshops, have the seats visibly removed but still reachable. After the workshop gets into flow state, people will start looking for seats only when they deserve a break.

Don't push it too forward: some people might have a real *need* for a seat. Don't let them suffer just because I told you so.

Reverse Narrative

(Pattern)

[FIXME: complete this one]

The Right To Be Wrong

(Pattern)

We are exploring, so being wrong is a legitimate state. Being *visibly wrong* is even better, so we can find somebody in the room with the information we need in order to learn something valuable. [FIXME: complete this one]

Single Out the Alpha-male

(Pattern)

Slack day after

(Pattern)

The workshop is essentially discovery and opening Pandora's Boxes. The outcome of this discovery is unknown. The necessary actions are unknown too, but we'll be targeting the most relevant blocker in the business flow.

Ideas and conversation emerging from the workshop will sparkle the urgency to do something about the most compelling problem in the whole flow.

Moreover, the workshop might favor extroverts over introverts who might need a quieter arrangement in order to formalize their thoughts.

therefore

Reserve a slack time after the workshop, to immediately start attacking the real problem. It can be used to start coding a prototype or sketching a candidate solution, or calling another meeting with the appropriate persons.

Nothing is more annoying that having a clear and definite vision of what needs to be done *now* and being reserved for doing something else instead.

Speaking out loud

(Pattern)

Our brain is a lazy machine. It will try to skip deep analytical thoughts in favor of quick pattern matching whenever possible. In practice - especially when we're running out of energies - it will take shortcuts and *pretend we understand something even if we don't*[^HSLDM].

:[^HSLDM] Like in the classic Homer Simpson vs the Lie Detector Machine: https://www.youtube.com/watch?v=k9Ylu0YywoA

Like most of our brain's loopholes, the problem is that we won't be aware of our brain taking a shortcut. Or worse: we might be annoyed by this pedantic need of precision on a trivial issue and call a *"Come on! Let's not waste our time on this one!"*

therefore

If you find yourself indulging in shortcuts, maybe a break will be necessary, but a great trick to detect inconsistencies is to exercise the verbal part of your brain by speaking out loud. I mean loud enough so that you form complete sentences that you peers are supposed to hear.

The moment you do that, you'll also exercise the part of your brain whose task is to avoid to look or *sound* stupid in public, or to lie unintentionally. You'll be surprised how many corrections and more specific tricks will be triggered by this practice.

Start from the center

(Pattern)

[FIXME: complete this one] See also Start from the beginning and Start from the extremes

Start from the extremes

(Pattern)

Exploration without a clear scope limitation works fine in Big Picture workshop where the goal is massive learning and risk discovery, but isn't a good fit for design sessions tailored around a specific use case or a Minimum Viable Product.

therefore

Unlimited Modeling Surface

(Pattern)

Visible Legend

(Pattern)

What are we doing? What was the meaning of *that lilac sticky note*? Should I use a *blue* or *orange* sticky here? Whenever people think questions like this, they are *distracted from their goal*. Modeling a complex business flow is difficult enough without the extra clutter of *having to remember the notation*.

If we're using incremental notation we could actually make things *worse*. If every new bit of notation is the result of an agreement on the fly between participants, the amount of *cognitive load* could be overwhelming.

therefore

Set up a *Visible Legend* available to all participants. Make it vivid and easily readable. Don't use symbols: you have the very same modeling tools available. You can use an orange sticky note and name it 'Domain Event', same with hot spots and so on. [FIXME: example image] Basically, use every trick to keep the cognitive load as low as possible.

With incremental notation this means that the facilitator will be creating the legend on-the-fly. Having a prepared *all-in-one* legend could save some time, but will also make it harder to steer the workshop reacting to the participants' mood and interests.

37. Anti-patterns

This is probably going to be one of the most awkward sections in the book, since most of the items listed here will sound like *plain old common sense*.

The problem is *common sense works only to a given extent*. Just like the abuse of antibiotics ended up increasing the probability of antibiotic-resistant diseases, the abuse of common sense is behind most entangled dysfunctions in complex environments, like …your workplace, for example.

So here it is, our little list of dysfunctional applications of what looks like plain old common sense.

Ask Questions First

(Anti-Pattern)

Every group of people starting an EventStorming workshop always starts self-organizing into *asking questions* to one person, usually the domain expert. It's so obvious that we don't even realize that this approach is totally sub-optimal for many reasons:

- one person is asking, one person is answering, many people are just klistening, or waiting their turn;
- only one topic at a time (serialization and low throughput);
- high risk of digging into one single topic without exploring the whole thing;
- high risk of spending a lot of time just stating the obvious (and *boring* is bad);

The best way to overcome this problem is to steer the workshop into massive parallel writing first, but to achieve that we need to establish the right to be wrong.

However, key questions are not forbidden, only *postponed*. They will spontaneously arise in front of some sticky note that doesn't resonate with the domain expert daily experience.

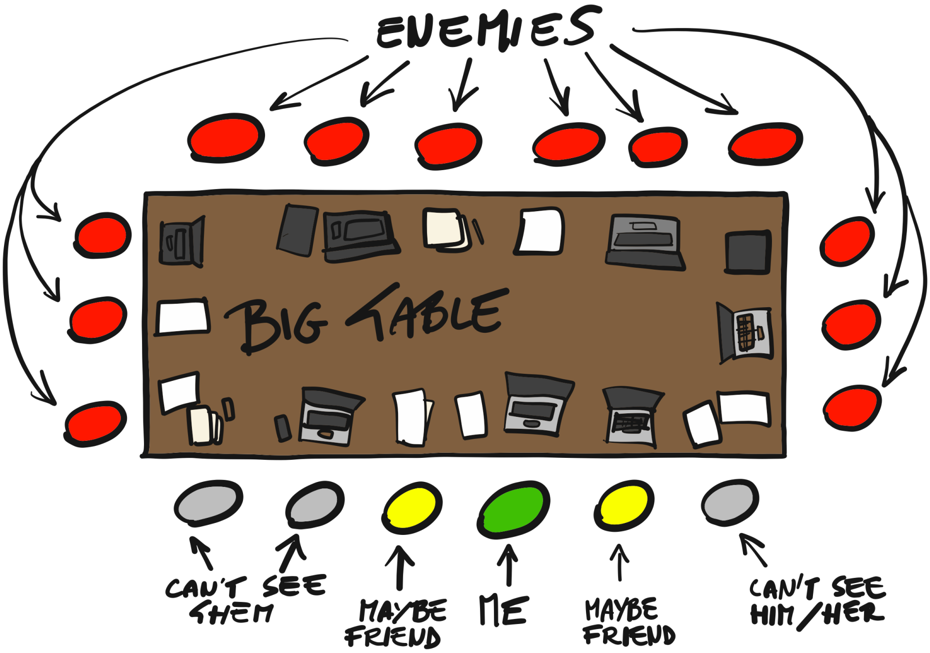
Big Table at the center of the room

(Anti-Pattern)

: Every large company has a meeting room with a huge table at the center. They may be ludicrously expensive, but …hey! Who cares, it's a *status* thing. We'll save the money on programmers monitors[1].

---

[1] Yes, I am ranting. Sorry about that.



Big tables are forcing you to choose a seat, defining upfront who's going to be your friend or foe during the meeting

Committee

(Anti-Pattern)

A common reaction to the *horror vacui*, the fear of an empty whiteboard at the beginning of a workshop is to form small groups and start reaching an agreement about what to write on our stickies.

This is reasonable and *damn wrong* at the same time, so here's a list of reasons why you should try to break up committees as soon as you spot them.

- Reaching an agreement is **expensive**: it requires consensus by every party involved, and this may take a huge amount of time.
- An agreement **limits the team bandwidth** to one topic, breaking committees allows for a higher parallelism an more collective throughput.
- Committees are **filtering out options**: they choose one wording for a given sticky note, and make the discarded ones invisible (see also make some noise).
- Committees fall into **confirmation bias**: forcing an agreement inevitably leads to re-framing the problem in *"the usual way"*. If we're looking at the problem from a different perspective, our worst mistake might just to discover the same old things.

In general, it is fun to see how the workshop speed dramatically increases after breaking up committees. But you can only let people fail and change the rules: explaining this in advance usually doesn't help much.

Divide and Conquer

(Anti-Pattern)

Many people in the software development field forgot the origin of the latin *divide et impera* that referred to a strategy to keep enemies divided. That was a strategy to maintain the *status quo*, not to solve big problems.

However, while decomposing a problems in smaller one is usually a good strategy for writing software, it doesn't mean it is a good strategy for everything.

The problem with this approach is that it tends to artificially reduce the exploration scope of a problem with sentences like *"Let's focus only on this part"*. *Focusing* is another verb with a default positive connotation, so it's shouldn't be wrong. However, the moment we *focus* we're deliberately excluding some information from the picture.

Focusing or *narrowing the scope* are often necessary choices when we're not provided a modeling space big enough to contain our problem, or when we can’t have all the key people together. In EventStorming we try to make the opposite possible allowing large scope exploration, and a Conquer first, divide later approach.

Do the right thing

(Anti-Pattern)

Dungeon Master

(Anti-Pattern)

Follow the leader

(Anti-Pattern)

Human Bottleneck

(Anti-Pattern)

Karaoke Singer

(Anti-Pattern)

[FIXME: This needs the hardest picture of all]

Some people like to dominate the discussion. Some people like the sound of their own voice *a lot*. Some people need to make it look like their team is helpless without them.

unfortunately

Actually, it doesn't have to be a discussion. The team doesn't have to reach any agreement (especially in Big Picture EventStorming), and slowing down the team in order to reach an agreement on any single sticky note, before writing is actually the most wasteful approach:

- we'll waste *time* by reducing throughput to only one person;
- we'll waste *energy* trying to reach consensus;
- we'll waste *insights* from people with a diverging idea, maybe not worth the effort.

therefore

Look closely to the emerging team dynamics. Point out that *consensus* is not necessary and diverging voices are actually valuable (see also Make Some Noise).

Make it clear that we're not looking for precision immediately and that every participant has The Right To Be Wrong.

Make sure that every participant has everything needed in order to participate independently (see One Man One Marker).

Precise Notation

(Anti-Pattern)

Every discipline has some specific notation. A precise standard notation allows more sophisticated conversations between experts, and allows for more precise modeling when used inside specific drawing tools.

unfortunately

Not every conversation is *expert-to-expert* or *expert-to-tool*. In fact the most significant conversation in software development are the ones between people that master the notation and the others that don't. Forcing people to adopt a specific language to join a conversation is no different from saying *"We're going to discuss something very important here, and we're going to discuss it in mandarin Chinese. Feel free to join the conversation!"*

The unwanted effect of sticking to a precise notation is to exclude key people from the conversation. Whenever the conversation becomes *specialist-to-specialist*, the one who is not familiar with the used jargon gets in a position of disadvantage when it comes to understanding, or correcting the model.

therefore

We'll try to avoid specific notation *during the workshops*. The quality of the conversation is what we should concentrate our attention. If notation gets in the way, then we should simplify it to the point that this is no longer a problem. See also Grounded Notation and [Visible Legend].

This doesn't mean that we shouldn't capture the information in a more precise notation later. In fact, a working prototype is another form of model in a *machine-readable* specific notation.

Religion War

(Anti-Pattern)

When two or more people have already figured out a model, or a solution, they might be tempted to skip the details and jump straight into the discussion, before having laid out the details.

unfortunately

My obvious is different from your obvious, and from everyone else's obvious too. But whenever something is invisible, and one has to explain it to someone else, discussions get abstract and start sucking time, backfiring into an endless old-style discussion.

therefore

Go visible as quickly as possible. Enforce

we only talk about visible things

as a modeling agreement between all parties. Break abstract discussion, by asking to write the missing building blocks.

Often you'll need to add more space to allow both solutions to be visible. In fact, if the space is not large enough to contain two alternatives, you are implicitly signaling that the team will have to choose a solution *upfront*.

Only choose between visible alternatives. Once both solutions are visible, the difference will probably be obvious to everybody.

Maybe you'll find out that the difference is negligible, and wasn't worth the standoff. Or you'll discover that it is significant, but now you are in the position to easily challenge both the alternatives with different scenarios, and discuss their probability and business significance.

Once the two solutions are visible the choice isn't limited to A or B. People could start contaminate the proposal with more alternatives, or merging the good parts anding up with something better.

Whatever would be your final preference, you'll have a better understanding of the reason why a solution has been chosen. And of the alternatives available in case it turns out the chosen solution isn't as good as it looked like.

The Spoiler

(Anti-Pattern)

“What’s the point of exploring together something that is already clear in my mind? Haven’t we already talked about it? I even wrote a specification document, just read through it and don’t waste my time with pointless questions!”

Often, there’ll be a person that knows a lot of the story. Maybe the person that developed the previous solution (see also the Dungeon Master on this specific case), or maybe the business analyst that already did dome exploration on the current matter. The more work they did, the less they’ll be inclined to repeat it: doing things twice always looks like waste, and they might be bored to repeat the same things over and over.

Things can get worse if the exploration phase led to some form of specification document. Since the company put some money on that one, it has to deliver some value, hence people will have to read it.

unfortunately

It’s not the expert knowledge that really matters. As we already mentioned [FIXME: where?] it’s developer understanding that gets captured in code and released in production. And developers need to *learn*.

But a specification document, or a person telling the distilled knowledge isn’t necessarily the most efficient way to learn on a complex matter. Sometimes, intermediate steps are necessary, sometimes *making mistakes is necessary* in order to achieve deeper learning.

Moreover, specification documents are an incredibly *boring* way to convey any form of knowledge. And we know that boredom is the arch-enemy of learning. Or it may shortcut all the thinking process necessary in order to understand why things are in a given why and why it matters. Distilled knowledge will turn into a dogma, leaving little space to maneuver in order to change them.

therefore

Forget the specification document. Make clear that we'll need to improve developers understanding, and that this will be faster and more efficient in an interactive workshop.

EventStorming doesn't try to make things right from the beginning. In fact we'll be progressing towards a common model, by exposing all the wrong models at once. Unexperienced colleagues have the right to place wrong sticky notes on the modeling surface, and this will trigger a meaningful conversation by somebody that knows the real story.

Start from the beginning

(Anti-Pattern)

Where should we start? Many times, you'll find somebody that *starts from the beginning* by placing the first domain event right at the leftmost edge of the modeling surface. There are two major pitfalls here:

1. In general, this is a very unwise usage of the available space: we have a huge surface and we're just constraining expansion to happen only in one direction.
2. We are implicitly signaling other people that there's nothing to explore *before the starting point*.

[FIXME: image here]

The godfather

Have you ever been in a meeting where, despite having 12 people in the room, only a few of them were entitled to speak? I was, and it was the only time in my career when I was ready to cancel the whole workshop and send everybody home.

38. RED ZONE

Event the best facilitators have weak points. I am not among them, so I have many weak points. I rationalized the thing by repeating me that facilitation is not a 100% guaranteed result process.

In real life, there are mutual selection processes in place to be sure that there's a good match between a client organization and a consultant, facilitator.

Fresh Catering

(Emergency Exit)

Providential Toilet Door Malfunctioning

(Emergency Exit)

Specific Formats

A brief recap of what is needed for each workshop format, in recipe style.

39. Big Picture EventStorming

Ingredients:

- A large meeting room
- Huge modelling surface (50 metres paper roll)
- People with questions
- People with answers
- A facilitator
- Many flipchart markers (one for each person plus 3 backup)
- Plenty of sticky notes:
  - Orange (500 squared 76x76)
  - Blue (200)
  - Purple (200)
  - Pink (100 rectangular)
  - Yellow (200 small rectangular)
- *Just in case:*
  - Green (100 squared 76x76)
  - Pale Yellow (100 rectangular)

Prepare the room in the standard way: table and seats aside. Keep the seats available for later use.

Take with you at least 50 metres of plotter paper roll: with *big picture exploration* you don't know how big the model is going to be in advance. Get somebody to help you setup the room. If everything's ready before the meeting start, you probably already made an impression.

Provide refreshments. After a while, the brains will start to work at full speed, so something like fresh fruit will be necessary. Don't rely on heavy food or industrial junk food. Not today.

The organizer should provide a little introduction: there’s no need to explain EventStorming, like “we’re going to do this and then this” [FIXME: This will go in the facilitator’s chapter] but to clearly state the workshop goals and to give participants *confidence* that you know what you’re doing with their time. Remember: *people are there for* ***their business goal****, not for EventStorming*.

40. Design-Level EventStorming

Ingredients:

- A large meeting room
- Huge modelling surface (50 metres paper roll)
- a completed Big Picture EventStorming
- a clear focus on a given business process or key feature
- the development team
- the relevant domain experts
- A facilitator
- Many flipchart markers (one for each person plus 3 backup)
- Plenty of sticky notes:
  - Orange (300 squared 76x76)
  - Blue (300)
  - Purple (200)
  - Lilac (200)
  - Pink (100 rectangular)
  - Yellow (200 small rectangular)
  - Green (200 squared 76x76)
  - Pale Yellow (100 rectangular)

Prepare the room in the standard way: table and seats aside. Keep the seats available for later use.

In this scenario we won't be stopped by timeout: the completion criteria is mostly the level of confidence with the modelled solution.

Next actions

Just start coding a prototype as soon as possible. Get some good food, but turn the exploration into working code, and write down the resulting questions.

41. Value Stream EventStorming

Ingredients:

- A large meeting room
- Huge modelling surface (50 metres paper roll)
- People with questions
- People with answers
- A facilitator
- Many flipchart markers (one for each person plus 3 backup)
- Plenty of sticky notes:
  - Orange (500 squared 76x76)
  - Blue (200)
  - Purple (200)
  - Pink (100 rectangular)
  - White cardboard (100 A7 format)
  - Green (100 squared 76x76)
  - Yellow Small (300)
- *Just in case:*
  - Pale Yellow (100 rectangular)

Glossary - 40% (but do you really care?)

Writer's note

I still have no idea whether this should be a simple reference for terms or something more structured, with deeper explanation. A ***Modeling Building Blocks*** *information is needed, but it's probably better to localize the explanation where is needed, and summarize it here.*

Fuzzy by design

Here is another chapter where I am going to intentionally disappoint many readers. If you're looking for a definition of the terms used throughout the book, here you are! But if you're looking for a *precise* definition, you won't find it.

It's not that I am lazy, or messy. I am, but that's not the point. The real reason is that in many situations, *precision gets in the way*. A large portion of EventStorming is about making significant conversations possible, and if we have to know what an *Aggregate* is in order to speak, well… that isn't going to work!

Actor

Aggregate
: Aggregate is one of the fundamental patterns described in Domain-Driven Design, Tackling Complexity at the Heart of Software.

:An aggregate is a *unit of consistency* within the Domain Model: a group of

Bounded Context

[FIXME: …]

Command

[…]

CQRS - Command-Query Responsibility Segregation

an architectural style enforcing strong separation between *Commands* actions that trigger a system behavior, and *Queries* that only access data. [FIXME: more info]

Domain Event:

In Domain-Driven Design, the term refers to a software design pattern, useful for modeling event-based systems and processes.

The term was originally defined on Martin Fowler's Bliki in 2005: http://martinfowler.com/eaaDev/DomainEvent.html and become popular once Greg Young started pushing the idea of using Event Sourcing and CQRS to model complex high-performance enterprise systems.

In *EventStorming* we are not looking for *precision*, but for meaningful conversations, that could engage the vastest company of stakeholders. So, a Domain Event is an *orange sticky note with a verb at past tense*, referring to something that happened in the domain.

[FIXME: picture a domain event. Not a domain event]

Event-Based Modeling

the original name for the EventStorming workshop, originally presented at the Italian Agile Day 2012 in Milan. Accompanying slides are available at â€¦.

Event-Driven Architecture

[…]

Event model
the physical outcome of an event storming session, be it the paper roll, a picture of it or its translation into a digital support.

Event Sourcing :

EventStorming
Go back to page one and re-read the book.

Hypocrite Modeling
when you're modeling a system with strict validation rules that cannot be fulfilled in the real world, sort of no-sex-before-marriage, and everybody finds a way to cheat.

Impact Mapping

Minimum Viable Product
[FIXME: …]

Model Storming
The process that unconsciously led me to EventStorming. Unfortunately I didn't realize it during the process, so I am responsible for making a bit of a mess. However, while EventStorming is the somewhat codified way to model business processes collaboratively, *Model Storming is the meta-process that let's you collaboratively model virtually everything* without having an idea of how it will look like at the end.

Persona In user-centered design and marketing, *personas* are fictional characters created to represent the different user types that might use a site, brand, or product in a similar way.

Source: Wikipedia[1] quoting William Lidwell; Kritina Holden; Jill Butler (1 January 2010), *Universal Principles of Design*, Rockport Publishers, p. 182, ISBN 978-1-61058-065-6

---

[1] https://en.wikipedia.org/wiki/Persona_(user_experience)

Read Model

a data-oriented model, with no specific behavior, which is normally tailored around a specific use case for a specific user.

Theory of Constraints

as explained by Eliyahu Goldratt in “The Goal”, ToC actively focuses on the search for the main system constraint. In a given flow there’s normally one main constraint, usually called *‘the bottleneck’*: improving the behavior around it results in major system improvements, while improving somewhere else leads to negligible results or even worsen the system performance.

User Story

User Story Mapping

Tools

A lot of the conversation revolving EventStorming and visual facilitation is about *“which markers do I use?”*. So here is a sort of dedicated F.A.Q. to the topic. Disclaimer: I added some link to the producers website whenever possible. I am not affiliated with any of those producers. Even more: most of the websites listed here suck big time, so my apologies if you get trapped into inconsistent navigation, ridiculous user experience and so on.

Modeling Surfaces

Paper Roll

The iconic paper roll.

I may adopt different strategies, given the context. Here are some of the most common choices.

- **Guerrilla Workshop**: I might be running one on the fly, I don't have so much time for the preparation. In this case my first choice is the IKEA Måla paper roll (it can be found in the kids area). The paper is cheap and somewhat yellowish, and the width is limited, so you'll surely need a double-decker or more. But it fits in my backpack, so I can carry it with me in a *just-in-case* fashion. The whole thing started exactly in this way.

- **Prepared workshop**: in this case, I prefer relying on more professional plotter paper roll. It comes in different format, so I may choose different options here. If I am traveling by car I may be taking a 90cm width roll with me, while if I am traveling by plane the maximum I was able to fit into my trolley was a 60 cm one.

Can we live without it?

Well, the paper roll has never been mandatory: it became the icon of EventStorming because in the largest majority of workplaces the idea of an unlimited modeling surface is an illusion.

Writable walls

In order to walk the path[FIXME: or is it walk the walk?] according to our principles, in our company every wall is writable. We applied a special paint on top of the existing one

Markers

You should already be familiar with the **One Man One Marker** rule. However, given the workshop constraints you'll might need more than one type of markers, depending on the use.

Writing on whiteboards (or erasable surfaces)

Writing on stickies

Here is where the normal whiteboard marker is simply too big to allow us to write anything significant, while a normal pen isn't visible enough. My favorite choice here is the **BIC marking pocket 1445**[2] or the **Sharpie Fine Point permanent marker**[3].

Writing on flip charts

Normally, writing on the flip-chart is reserved only to the facilitator, but it might not necessarily be a rule. To write on a flip-chart, a larger font size will be needed and a larger marker too. If you're a beginner a round tip marker will do the job really easily. If you're visual scribing like a pro, then a *chisel tip* will allow for better results. [FIXME: links and maybe an image]

Stickies

The glue is the most important thing: you just don't want to invite your company big guys to a workshop and waste their time picking up falling stickies from the floor. So don't pretend you're saving money and choose something with a decent glue.

---

[2] http://www.bicworld.com/en/products/details/103/marking-pocket-1445

[3] http://www.sharpie.com/it-IT/sharpie-products/marcatori-permanenti/sharpie-fine-point-sp-00013--4

A reasonable supply of orange stickies.

I am loyal to [3M super sticky]

Removable labels

The great thing about sticky notes is that they can be moved around. When people start labeling areas for Subdomains or Bounded Contexts they tend to write directly on the surface. But if the surface is paper, this is irreversible: can't rename it or move it without making a mess.

My not so secret trick is to use a labeling and covering tape for this. It's just a white sticky note that I can rewrite or move around whenever the merging structure calls for it.

Labeling and covering tape

Static pads

These are lovely on a smooth surface.

Recording results

Bibliography

Agile Retrospectives - Esther Derby, Diana Larsen
Apparently focused on Agile retrospective only, this book is a goldmine for suggestions about how to run workshop, or - more generally - how to handle situation where you put a lot of clashing personalities in the same room.

Agile Software Development, the Cooperative Game - Alistair Cockburn
One of the first book I read about agile software development, and a game-changer for me. It opened my eyes towards the idea of Information radiators and around communication as one of the major bottleneck in software development.

Analysis Patterns - Martin Fowler
In this book, Martin explores recurring and repeatable patterns for modeling specific portions of a business domain. Although it's somewhat outdated (pictures are in a pre-UML notation) it's definitely a worthwhile read for a sophisticated modeler

Antifragile - Nassim Nicholas Taleb
Fragile systems get damaged by volatility and disorder. Antifragile systems have the ability to take advantage from disorder.

Black Box Thinking - Matthew Syed
From covering up mistakes, to cognitive dissonance, to what makes an organization learn and be a sustainable place to work, Mathew Syed manages to connect an impressive amount of dots in a meaningful and readable way.

Brain Rules - John Medina
A great book for understanding the basic mechanism of our brain, and a handful of practical advice about how to take the best out of it.

The Connected Company - Dave Gray
Hierarchy used to be the canonical form for every organization structure. However, it's not necessarily the most efficient, depending on the organization's purpose. In this book, Dave Grays collects and describes several examples of organizations and companies that succeeded with different organizational structures.

Design Patterns, [FIXME and something else] - Gang of Four [FIXME: the actual names]
[FIXME say something smart here]

Domain-Driven Design, Tackling Complexity at the Heart of Software - Eric Evans
This is the book that started everything, and that still surprises me. After many years I still find pearls of wisdom entangled in Eric's software modeling approach. One among many: we are knowledge crunchers, EventStorming is just my attempt to make this statement real.

Drive, the surprising truth about what motivates us - Daniel H. Pink
[FIXME: tell the story]

eXtreme Programming Explained - Kent Beck
[FIXME say something smart here]. Interestingly, among the agile practices, Kent places 'Build a system metaphor', which I guess is exactly what EventStorming is all about.

GameStorming - Dave Gray
*If you wonder how it could be possible to turn boring unproductive meetings into incredibly engaging and effective ones, this is the book for you. Dave will provide information about the basic structure of a *storming meeting, and specific tips to use in the different phases.*

Getting to Yes - Roger Fisher, William Ury
A small and popular negotiation book. If you're surprised to find a negotiation book in the references, then you probably need to think twice about what collaborative design really is.

Growing Object Oriented Software, guided by tests - Steve Freeman & Nat Pryce

The best book about implementing a nontrivial software system applying Test-Driven Development principles in the small and in the large. It's also the book that made the concept of walking skeleton popular among software developers.

Impact Mapping - Gojko Adzic

A great book about turning the discussion into software requirements into a discussion about doing the right thing. Writing software is expensive, but many organizations keep pushing for writing more. Why? What is the outcome they seek? Is it "more software"? Or more likely they're looking for an impact on user behavior, or on company revenues? If that's the reason, what is the shortest and more effective path to achieve this result?

Implementing Domain-Driven Design - Vaughn Vernon

Introducing Deliberate Discovery - Dan North

This is one of the best explanation why learning is the real bottleneck in software development. The whole article can be read here[4].

Kanban - David J. Anderson

The book that defined Kanban (capital 'K') software development process, by putting together Agile, Lean and Queue Theory principles.

Management 3.0 - Jurgen Appelo

Jurgen's book opened a whole new universe to me. It connected topics and disciplines that have been separated for ages and provided a consistent view of the whole. Even more, he made the whole thing look easy. I still don't know how he managed it.

Patterns of Enterprise Application Architecture - Martin Fowler

[FIXME say something smart here]

---

[4] http://dannorth.net/2010/08/30/introducing-deliberate-discovery/

Pragmatic thinking and Learning - Andy Hunt

An introduction to how the brain works, written by a software developer. One of the best entry point into the magic realm of the human brain.

Switch - Chip & Dan Heath

One of the easiest to read book about change management. But also a great collection of wisdom about how little constraints can influence people's behavior in unexpected ways.

The Goal - Eliyahu Goldratt

Written as a novel, the book illustrates the principles of Theory of Constraints, giving a very crucial significance to the concepts of Bottleneck and throughput.

The Lean Startup - Eric Ries

Probably the most influential book in the startup world, and more. The idea of validated learning is incredibly powerful and drives software development in a way that makes traditional management and planning look like stone age, especially in exploratory domains.

Thinking Fast and Slow - Daniel Kahneman

This is an amazing book explaining how the brain works in many circumstances. From managing risks and probabilities to multitasking. If your work is brain-intensive (and I bet it is), then you have no excuses not to read this one. You'll probably find many logical explanations to real world situation in your daily and working life. You can't change people's brains, including yours. So it's better to know how it really works to find strategies around it instead of against it.

This is Lean - Niklas Modig, Pär Åhlström

A great entry point to lean thinking, with some notable stories that will introduce the reader into the mindset. Very well written and very clear, with some really valuable advices when it comes to choose a corporate strategy according to principles.

User Story Mapping - Jeff Patton

Story Mapping has been one of the key sources of inspiration for EventStorming, and a lot of the thinking behind it is the same. While reading the first chapters of Jeff's book I've found myself saying: "this is exactly what I wanted to write!", and it's also very well put. If you're struggling to put a product vision into practice, this is the book for you.

Visual Meetings - David Sibbett

David Sibbett made the profession of Visual Facilitator visible, showing the value of visually supporting large scale meetings, and providing precious advice. The book is great but the single sentence "Can't do system thinking without visualization" is already worth the price.